教育部高等学校网络空间安全专业教学指导委员会
中 国 计 算 机 学 会 教 育 专 业 委 员 会 共同指导

奇安信
新 一 代 网 络 安 全 领 军 者

奇安信集团组织编写

网络空间安全重点规划丛书

顾问委员会主任：沈昌祥 编委会主任：封化民

代码安全

杨东晓 章磊 付威 金竹君 编著

Cyberspace Security

根据教育部高等学校信息安全专业教学指导委员会编制的
《高等学校信息安全专业指导性专业规范》组织编写

清华大学出版社

网络空间安全重点规划丛书

代码安全

杨东晓　章　磊　付　威　金竹君　编著

清华大学出版社
北　京

内 容 简 介

本书共分为9章。系统地介绍了软件安全开发的基础知识，包括软件开发的现状和常见的开发生命周期模型；在传统需求分析方法基础上介绍软件安全需求与设计，重点介绍威胁建模方法；结合软件开发模型介绍软件发布和部署阶段的安全措施；具体结合C、C++、Java、PHP、Python等语言，详细讲解其安全现状、常见漏洞和编码规范；概述了安全测试的流程和方法，并结合典型案例让读者能够对这些代码中的安全漏洞有比较全面和深入的了解，从而在实际开发过程中避免出现类似漏洞。

本书既可作为高校网络空间安全、信息安全等相关专业的教材及网络工程、计算机技术应用培训教材，也可作为负责网络安全运维的网络管理人员和对网络空间安全感兴趣的读者的基础读物。

图书在版编目(CIP)数据

代码安全/杨东晓等编著. —北京：清华大学出版社，2020.4
(网络空间安全重点规划丛书)
ISBN 978-7-302-55090-7

Ⅰ.①代…　Ⅱ.①杨…　Ⅲ.①软件开发—安全技术　Ⅳ.①TP311.522

中国版本图书馆CIP数据核字(2020)第047315号

责任编辑：张　民　战晓雷
封面设计：常雪影
责任校对：时翠兰
责任印制：杨　艳

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦A座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-83470236
印 刷 者：北京富博印刷有限公司
装 订 者：北京市密云县京文制本装订厂
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：10.25　　**字　　数**：234千字
版　　次：2020年7月第1版　　**印　　次**：2020年7月第1次印刷
定　　价：35.00元

产品编号：085315-01

网络空间安全重点规划丛书

编审委员会

出版说明

21 世纪是信息时代，信息已成为社会发展的重要战略资源，社会的信息化已成为当今世界发展的潮流和核心，而信息安全在信息社会中将扮演极为重要的角色，它会直接关系到国家安全、企业经营和人们的日常生活。随着信息安全产业的快速发展，全球对信息安全人才的需求量不断增加，但我国目前信息安全人才极度匮乏，远远不能满足金融、商业、公安、军事和政府等部门的需求。要解决供需矛盾，必须加快信息安全人才的培养，以满足社会对信息安全人才的需求。为此，教育部继 2001 年批准在武汉大学开设信息安全本科专业之后，又批准了多所高等院校设立信息安全本科专业，而且许多高校和科研院所已设立了信息安全方向的具有硕士和博士学位授予权的学科点。

信息安全是计算机、通信、物理、数学等领域的交叉学科，对于这一新兴学科的培养模式和课程设置，各高校普遍缺乏经验，因此中国计算机学会教育专业委员会和清华大学出版社联合主办了“信息安全专业教育教学研讨会”等一系列研讨活动，并成立了“高等院校信息安全专业系列教材”编审委员会，由我国信息安全领域著名专家肖国镇教授担任编委会主任，指导“高等院校信息安全专业系列教材”的编写工作。编委会本着研究先行的指导原则，认真研讨国内外高等院校信息安全专业的教学体系和课程设置，进行了大量具有前瞻性的研究工作，而且这种研究工作将随着我国信息安全专业的发展不断深入。系列教材的作者都是既在本专业领域有深厚的学术造诣，又在教学第一线有丰富的教学经验的学者、专家。

该系列教材是我国第一套专门针对信息安全专业的教材，其特点是：

① 体系完整、结构合理、内容先进。

② 适应面广：能够满足信息安全、计算机、通信工程等相关专业对信息安全领域课程的教材要求。

③ 立体配套：除主教材外，还配有多媒体电子教案、习题与实验指导等。

④ 版本更新及时，紧跟科学技术的新发展。

在全力做好本版教材，满足学生用书的基础上，还经由专家的推荐和审定，遴选了一批国外信息安全领域优秀的教材加入系列教材中，以进一步满足大家对外版书的需求。“高等院校信息安全专业系列教材”已于 2006 年年初正式列入普通高等教育“十一五”国家级教材规划。

2007 年 6 月，教育部高等学校信息安全类专业教学指导委员会成立大会

暨第一次会议在北京胜利召开。本次会议由教育部高等学校信息安全类专业教学指导委员会主任单位北京工业大学和北京电子科技学院主办，清华大学出版社协办。教育部高等学校信息安全类专业教学指导委员会的成立对我国信息安全专业的发展起到重要的指导和推动作用。2006 年教育部给武汉大学下达了“信息安全专业指导性专业规范研制”的教学科研项目。2007 年起该项目由教育部高等学校信息安全类专业教学指导委员会组织实施。在高教司和教指委的指导下，项目组团结一致，努力工作，克服困难，历时 5 年，制定出我国第一个信息安全专业指导性专业规范，于 2012 年年底通过经教育部高等教育司理工科教育处授权组织的专家组评审，并且已经得到武汉大学等许多高校的实际使用。2013 年，新一届教育部高等学校信息安全专业教学指导委员会成立。经组织审查和研究决定，2014 年以教育部高等学校信息安全专业教学指导委员会的名义正式发布《高等学校信息安全专业指导性专业规范》(由清华大学出版社正式出版)。

2015 年 6 月，国务院学位委员会、教育部出台增设“网络空间安全”为一级学科的决定，将高校培养网络空间安全人才提到新的高度。2016 年 6 月，中央网络安全和信息化领导小组办公室(下文简称中央网信办)、国家发展和改革委员会、教育部、科学技术部、工业和信息化部及人力资源和社会保障部六大部门联合发布《关于加强网络安全学科建设和人才培养的意见》(中网办发文〔2016〕4 号)。2019 年 6 月，教育部高等学校网络空间安全专业教学指导委员会召开成立大会。为贯彻落实《关于加强网络安全学科建设和人才培养的意见》，进一步深化高等教育教学改革，促进网络安全学科专业建设和人才培养，促进网络空间安全相关核心课程和教材建设，在教育部高等学校网络空间安全专业教学指导委员会和中央网信办资助的网络空间安全教材建设课题组的指导下，启动了“网络空间安全重点规划丛书”的工作，由教育部高等学校网络空间安全专业教学指导委员会秘书长封化民教授担任编委会主任。本规划丛书基于“高等院校信息安全专业系列教材”坚实的工作基础和成果、阵容强大的编审委员会和优秀的作者队伍，目前已经有多本图书获得教育部和中央网信办等机构评选的“普通高等教育本科国家级规划教材”“普通高等教育精品教材”“中国大学出版社图书奖”和“国家网络安全优秀教材奖”等多个奖项。

“网络空间安全重点规划丛书”将根据《高等学校信息安全专业指导性专业规范》(及后续版本)和相关教材建设课题组的研究成果不断更新和扩展，进一步体现科学性、系统性和新颖性，及时反映教学改革和课程建设的新成果，并随着我国网络空间安全学科的发展不断完善，力争为我国网络空间安全相关学科专业的本科和研究生教材建设、学术出版与人才培养做出更大的贡献。

我们的 E-mail 地址是：zhangm@tup.tsinghua.edu.cn，联系人：张民。

“网络空间安全重点规划丛书”编审委员会

前 言

没有网络安全，就没有国家安全；没有网络安全人才，就没有网络安全。

为了更多、更快、更好地培养网络安全人才，许多学校都加大投入，聘请优秀教师，招收优秀学生，建设一流的网络空间安全专业。

网络空间安全专业建设需要体系化的培养方案、系统化的专业教材和专业化的师资队伍。优秀教材是网络空间安全专业人才的关键。但是，这是一项十分艰巨的任务。原因有二：其一，网络空间安全的涉及面非常广，至少包括密码学、数学、计算机、通信工程等多门学科，因此，其知识体系庞杂、梳理困难；其二，网络空间安全的实践性很强，技术发展更新非常快，对环境和师资要求也很高。

“代码安全”是网络空间安全和信息安全专业的基础课程，通过介绍软件安全开发的模型和常见代码漏洞与解决办法，使学生掌握软件安全开发方法和常用代码编写规范。本书涉及的知识面宽，共分为 9 章。第 1 章介绍软件安全开发基础，第 2 章介绍软件安全需求与设计，第 3 章介绍 C 和 C++ 安全编码，第 4 章介绍 Java 安全编码，第 5 章介绍 PHP 安全编码，第 6 章介绍 Python 安全编码，第 7 章介绍软件安全测试，第 8 章介绍软件安全发布与部署，第 9 章介绍典型案例。

本书既适合作为高校网络空间安全、信息安全等专业的教材，也适合网络安全研究人员作为网络空间安全领域的入门基础读物。随着新技术的不断发展，作者今后会不断更新本书内容。

由于作者水平有限，书中难免存在疏漏和不妥之处，欢迎读者批评指正。

作　者

2019 年 12 月

目 录

第 1 章 软件安全开发基础

信息技术的广泛应用和网络空间的发展极大促进了经济社会繁荣进步，但同时也带来了新的安全风险和挑战。本章首先介绍软件安全开发的现状、问题和成因，其次阐述软件安全开发生命周期，介绍几种常见软件安全开发模型，使读者认识到软件安全开发的重要性，了解软件安全开发的基本思想和方法流程。

1.1 软件安全开发现状

1.1.1 软件安全面临的挑战

随着网络技术和应用的飞速发展，网络化和互联互通已经成为当前软件和信息系统发展的大势所趋，信息系统安全正面临着前所未有的挑战。一方面，信息系统与互联网或其他网络的互联增加，会导致系统受攻击面增大；另一方面，随着构建在信息系统之上的各种业务应用的不断丰富以及软件和信息系统复杂程度的不断提高，使系统面临的安全威胁空前增加；此外，代码行数的迅速增加，使系统中隐藏的各种安全隐患也随之增多，并且通常难以被发现和消除。

软件开发通常会引入缺陷。据 360 互联网安全中心的数据显示：普通软件工程师的缺陷密度一般为 50～250 个缺陷/千行源代码。由于有严格的软件开发质量管理机制和多重测试环节，成熟的软件公司的缺陷密度要低得多，普通软件开发公司的缺陷密度为 4～40 个缺陷/千行源代码，高水平的软件公司的缺陷密度为 2～4 个缺陷/千行源代码，而美国 NASA 的软件缺陷密度可低至 0.1 个缺陷/千行源代码，如表 1-1 所示。

表 1-1 代码缺陷密度

开发人员/组织类型	缺陷密度/(个缺陷·千行源代码$^{-1}$)
普通软件工程师	50～250
普通软件开发公司	4～40
高水平软件开发公司	2～4
美国 NASA	0.1
中国软件公司	6

根据奇安信集团多年的源代码缺陷测试实践统计，国产软件平均的缺陷密度为 6 个

缺陷/千行源代码，假设 1%的安全缺陷是可被黑客恶意利用以实施攻击的，则一个网银客户端大小的软件将可能存在 3～6 个可被利用的漏洞，由此可见国内软件安全形势的严峻。

为了保证新产品的面市时间，许多公司只注重软件开发而忽略了产品的安全问题，使得攻击者可以借助软件漏洞轻松入侵系统。国内外利用软件漏洞攻击信息系统导致的信息安全事件屡见不鲜，如图 1-1 所示。例如近年热门的共享单车，攻击者利用其软件漏洞，在短时间内就可对用户账户进行入侵，获取用户隐私信息，损害用户的合法权益；随着网络借贷平台的兴起，攻击者可利用平台框架代码中隐藏的漏洞入侵主机，进行非法借贷提现操作。人们已经逐渐意识到软件安全开发的重要性。



图 1-1　软件漏洞被利用的事例

面对如此严峻的软件安全开发现状，国家制定了相关法律法规和标准，积极推动网络环境安全建设。《国家信息安全等级保护管理方法》要求，对于自行开发软件的公司，应制定代码编写安全规范，开发人员应参照规范编写代码，在软件开发过程中必须对安全性进行测试，并在软件安装前对可能存在的恶意代码进行检测；对于从事外包软件开发的公司，应在软件交付前检测软件质量和其中可能存在的恶意代码。2017 年 6 月 1 日，《中华人民共和国网络安全法》正式实施，对业务系统安全审计提出了新的要求。《国务院办公厅关于印发政府网站发展指引的通知》中也提到，要对应用软件的代码进行安全分析和测试，识别并及时处理可能存在的恶意代码。

1.1.2　软件安全开发问题

在信息化建设过程中，软件系统的安全体系是一个复杂的体系，如果对系统软件、应用软件和第三方软件的安全问题以及开发、部署过程中的安全问题处理不当或者不加防范，都可能给整个系统带来巨大的灾难。

漏洞(vulnerability)是指软件中实际存在、能够导致系统崩溃或遭受攻击的安全问题。信息技术、信息产品、信息系统在需求、设计、实现、配置、运行等过程中,常会产生缺陷,这些缺陷以不同形式存在于信息系统的各个层次和环节之中。漏洞在多数情况下并不影响软件的正常功能,但如果被攻击者成功利用,就可能引起恶意代码的执行,对信息系统的安全造成损害,从而影响构建于信息系统之上的正常服务的运行,危及信息系统及信息的安全。

缺陷(weakness)通常是指从源代码层面分析出来的潜在安全隐患,它影响软件的正常功能,例如执行结果错误、页面显示错误等。在源代码层面进行缺陷检测和修复,能够大幅降低最终软件中的漏洞数量,起到"防患于未然"的作用,也是成本最低的方式。

错误(bug)是指计算机系统的硬件、系统软件(如操作系统)或应用软件(如文字处理软件)在运行过程中出错,并不局限于安全问题。

1. 常见的软件安全问题

常见的软件安全问题分为以下4种。

1) 系统软件安全问题

系统软件是指面向硬件或者开发者所设立的软件,是指控制和协调计算机及外部设备,支持应用软件开发和运行的系统。有代表性的系统软件有操作系统和数据库系统。操作系统是计算机系统的控制和管理中心,常见的操作系统有 Windows、Linux、UNIX、Mac OS 等。数据库系统是由数据库及其管理软件组成的系统,代表性的数据管理系统有 Oracle、MySQL 等。绝大多数系统和应用漏洞源于在软件开发的需求分析、总体设计和代码编制等过程中引入的缺陷。

近几年兴起的"破壳"(Shellshock)漏洞是系统软件安全问题的一个典型实例。这是一种典型的脚本注入漏洞,由于基于 Bash 的常用远程服务在输入过滤中没有严格限制边界,也没有作出参数合法化判断,攻击者可在 Linux 等服务器操作系统上远程执行任意命令。并且因为类 UNIX 系统是众多网络设备的底层操作系统,特别是 Bash 广泛地分布和存在于设备中,所以此漏洞的危害范围很大,涉及网络设备、网络安全设备、云计算和大数据中心等。

2) 应用软件安全问题

应用软件(application software)是和系统软件相对应的,是为满足用户不同领域、不同问题的应用需求而开发的软件,包括办公软件、多媒体软件和分析软件等。任何一种软件系统都存在脆弱性,应用软件漏洞可以看作编程语言的局限性。例如,一些程序如果接收到异常或者超长的数据和参数就会导致缓冲区溢出,这是因为很多软件在设计时忽略或者很少考虑安全性问题,且大多数开发人员缺乏安全培训或没有安全编程经验。常见的应用软件漏洞有 SQL 注入、跨站脚本攻击(XSS)漏洞、跨站请求伪造(CSRF)漏洞和爬虫盗链等。

近些年热门的 P2P 金融网站系统就是一个典型例子。随着互联网金融产业的兴盛,很多互联网企业通过购买通用软件模板或自主开发的方式建立 P2P 网贷平台。由于金

融网站开发普遍存在着安全技术力量不足的状况，网站系统中经常包含XSS漏洞、CSRF漏洞、平行越权漏洞、任意用户密码重置漏洞等开发漏洞。攻击者可以利用这些漏洞攻击网站后台，使得P2P平台遭遇系统瘫痪、数据被恶意泄露或篡改、资金被洗劫等安全问题。

3）第三方代码安全问题

为了加快软件开发的速度，大多数开发人员会使用第三方代码库帮助自己方便、快捷地实现想要的功能，例如拼写检查程序或图表制作工具。然而长期使用这样的第三方代码存在很大的安全隐患。如果某个库文件存在漏洞，且在程序开发设计阶段，开发者又忽略了第三方库代码的漏洞审查，那么所有使用了该库文件的软件程序都将面临安全威胁。这种漏洞大多属于突发性威胁，爆发速度快，波及范围广，值得高度关注。例如，OpenSSL中出现的心脏滴血漏洞（Heartbleed）、GNU Bash中出现的破壳漏洞（Shellshock）都属于这种漏洞。

OpenSSL是安全套接层密码库，同时还是一个第三方开放源代码软件包，Apache使用它加密HTTPS，OpenSSH使用它加密SSH，很多涉及资金交易的平台（如支付网站）也用它作为加密工具。2014年，OpenSSL中的心脏滴血安全漏洞被首次曝光，利用这个漏洞，攻击者不仅可以破解加密的信息，而且可以从内存中提取随机数据，即攻击者可以利用这个漏洞直接窃取目标用户的密码、私人密钥和其他敏感用户数据。由于大量软件都使用了存在漏洞的OpenSSL代码库，尤其是HTTPS网站（大多采用OpenSSL来实现对SSL协议的支持），致使国内外众多大型网站和厂商都受到严重影响。直到今天，很多设备上的OpenSSL漏洞仍然没有被修复。

4）新技术安全问题

在工业生产领域，随着计算机技术、通信技术、控制技术的发展和信息化与工业化的深度融合，传统的工业控制系统（Industrial Control System，ICS）逐渐向网络化方向转变，ICS面临的信息安全形势日益严峻。例如，著名的“网络超级武器”Stuxnet病毒有针对性地对伊朗布什尔核电站进行入侵，严重影响核反应堆的安全运营。

在日常生活中，智能手表、智能电视、智能冰箱、智能洗衣机乃至智能电饭煲等越来越多的新设备接入互联网，万物互联的时代已经开启，这些新型网络中的软件安全问题尤为引人关注。以车联网为例，德国安全专家发现宝马诸多车型的“联网驾驶”数字服务系统在安全性方面有瑕疵，攻击者可利用其中的漏洞在数分钟内以远程无线方式侵入车辆内部并解除车锁。其主要原因是汽车控制系统在验证解锁信号时只是向宝马服务器发送了一个简单的HTTP Get请求，并没有在传输过程使用SSL或TLS加密，致使攻击者可以轻松截获传输信息。

2 软件开发各阶段中的安全问题

在需求、设计和开发等软件开发流程中，如果在安全方面稍有不慎，就可能将安全漏洞带进软件中。绝大多数网络攻击都是针对和利用已知的、未打补丁的软件漏洞和不安全的软件配置进行的。

软件开发过程如图1-2所示，包括需求分析、设计、开发、测试、部署、运维6个阶段。

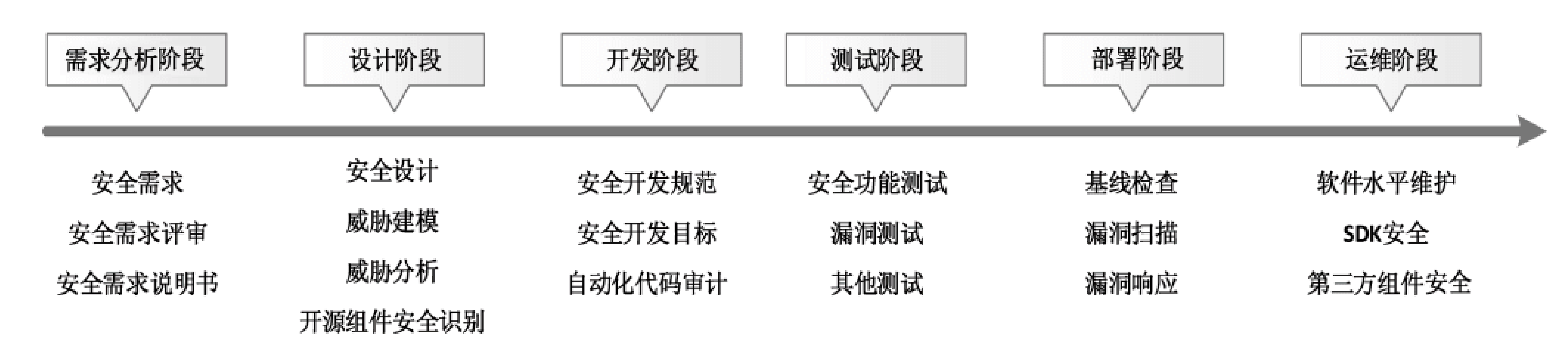


图 1-2　软件开发过程

在软件设计和开发的不同阶段，开发人员需要考虑不同的安全问题。

(1) 在需求分析阶段，大多数程序设计人员过多关注需要实现的功能，而忽视了安全需求，需求评审缺少安全方面的审计介入，安全目标不够明确。

(2) 在设计阶段，认证问题、授权问题、关键数据保护措施不够完善，身份认证和会话管理设置不当，第三方组件不安全等软件设计导致的漏洞层出不穷。例如，如果开发人员使用明文保存密码，就可能使得攻击者轻易获取用户的隐私信息，进而造成用户权益受损。CSRF、Direct Object Reference 等漏洞也属于软件设计缺陷。

(3) 在开发阶段，安全编码规范的制定、周期性检测、差距分析、修复跟踪等方面都较易出现漏洞，例如 SQL 注入、XSS 漏洞等，这会使得敏感信息易被泄露，攻击者可利用漏洞信息攻击服务器，盗取用户隐私信息，给企业和用户都造成严重损失。

(4) 在测试阶段，许多测试人员只关注软件的功能、性能和稳定性检测，而忽视源代码缺陷测试、黑盒测试、业务性安全测试和第三方组件安全评审等在安全开发中至关重要的测试步骤。

(5) 在部署阶段，缺少配置安全评审，系统的额外服务、端口安全，服务器默认用户、默认用例、权限过高的默认账户等方面都有可能造成软件缺陷。例如，默认口令会使得攻击者更容易获取管理员密码，从而入侵服务器，获取大量用户信息。

(6) 在运维阶段，软件水平是否完好无损，SDK 是否出现安全问题，软件本身是否出现安全问题，第三方组件是否产生安全漏洞，都是运维人员需要经常考虑的安全问题。

1.1.3　软件安全问题成因

现代软件功能强大而复杂，因此所有的软件都存在漏洞。只要软件存在漏洞，就有被利用的可能。许多公司为了保持竞争能力的领先地位，不断加快新产品的面市速度，却忽略了应用系统的安全问题，攻击者借助漏洞得以轻松入侵系统，窃取公司机密数据，使整个公司处于风险之中。事实上，几乎所有的网络攻击都是利用系统软件或应用软件中存在的安全缺陷实施的。为了保护国家、企业和个人的安全和利益，应该研究软件的安全开发方法。软件安全开发问题的产生原因有以下几点：

- 开源软件不安全。
- 项目任务工期紧张，未考虑安全编程的时间。
- 开发环境不安全。
- 开发团队缺乏必要的安全开发专业知识。
- 软件趋向大型化，第三方扩展增多。

1. 开源软件不安全

许多程序设计人员在开发软件时，更注重软件功能的实现，常使用大量未经安全检查的开源代码和公用模块，而缺少了对安全风险的管理。开源软件的来源一般有两类：一是开源机构，如 GitHub；二是营利机构，如 Red Hat。这些代码中可被利用的已知和未知缺陷较多。如果引用的开源代码本身存在漏洞，则所有使用该开源代码的软件都将陷入危机。

2. 项目任务工期紧张，未考虑安全编程的时间

软件开发公司工期紧、任务重，为争夺客户资源和抢夺市场份额，常仓促发布软件。软件开发人员为了在规定的时间内完成开发任务，往往只关注是否实现了所需功能，没有过多的时间和精力从攻击者的角度思考软件安全问题，对软件安全架构、安全防护措施认识不够。

3. 开发环境不安全

软件开发环境是用于支持软件的工程化开发和维护的一组软件。当开发环境出现安全问题时，使用该开发环境开发或维护的软件都将存在漏洞。例如苹果公司的 Xcode 开发工具，由于其体积过于庞大，通过苹果官方商店安装十分缓慢，于是很多开发者从网盘或迅雷等第三方渠道下载。攻击者利用这一点，将包含恶意功能的 Xcode 重新打包，发至各类苹果开发社区供开发者下载。来自各企业的开发者下载和安装包含恶意代码的 Xcode，用它来编写软件。这些被注入恶意功能的软件可以通过审核，在苹果商店上架，用户在下载和安装这些存在恶意代码的软件后，其系统也将被感染。

4. 开发团队缺乏必要的安全开发专业知识

在软件公司中，软件开发人员大多仅学会了编程技巧，很少受过安全能力与意识的培训，不了解安全漏洞的成因、技术原理与安全危害，由于其安全开发经验不足，不能更好地将软件安全需求、安全特性和编程方法相互结合以开发安全的软件；而项目开发管理者大多缺乏软件安全开发知识，不了解软件安全开发的管理流程和方法，缺少安全部署评审过程，不清楚安全开发过程中使用的各类方法和思想。实施软件的安全开发过程，需要开发团队所有的成员以及项目管理者都具备较多的安全知识。

5. 软件趋向大型化，第三方扩展增多

现代软件功能越来越强，功能组件越来越多且越来越复杂，操作系统的代码量也迅速增加。现在基于网络的应用系统更多地采用了分布式、集群和可扩展架构，软件应用向可扩展化方向发展，成熟的软件也可以接受开发者或第三方的扩展，使系统功能得到扩充。例如，Firefox 和 Chrome 浏览器支持第三方插件，Windows 操作系统支持动态加载第三方驱动程序，Word 和 Excel 等软件支持第三方脚本和组件运行，等等。然而这些可扩展性在增加软件功能的同时，也加剧了软件的安全问题。研究显示，软件漏洞的增长同软件复杂性、代码行数的增长成正比，即“代码行越多，缺陷也就越多”。

1.2　软件开发生命周期

软件开发生命周期(Software Development Life Cycle,SDLC)又称为软件生存周期或系统开发生命周期,是软件从产生直到报废的生命周期,其中包括问题定义、可行性分析、总体描述、系统设计、编码、调试和测试、验收与运行、维护升级、废弃等阶段,这种按时间划分阶段的思想方法是软件工程中的一种思想原则,即按部就班、逐步推进,每个阶段都要有定义、工作、审查,最终形成文档以供交流或备查,以提高软件的质量。

按照软件的生命周期,软件开发不再仅仅强调编码,而是概括了软件开发的全过程。软件工程要求每一阶段工作的开始必须以前一个阶段结果正确为前提,因此每一阶段都是按“活动—结果—审核—活动……”循环往复进行的,直至结果正确。

从概念提出的那一刻开始,软件产品就进入了软件生命周期。在经历需求分析、设计、实现、部署后,软件将被投入使用并进入维护阶段,直到最后由于缺少维护费用而逐渐废弃。这样的一个过程称为生命周期模型(life cycle model)。

典型的生命周期模型包括瀑布模型、迭代模型、增量模型、快速原型模型、螺旋模型、净室模型等。

1. 瀑布模型

瀑布模型(waterfall model)是最早出现的软件开发模型,它提供了软件开发的基本框架,直到现在仍然是软件工程中应用最广泛的过程模型。其核心思想是按工序将问题简化,将功能的实现与设计分开,便于分工协作,即采用结构化的分析和设计方法将逻辑实现与物理实现分开。

瀑布模型如图 1-3 所示,它将软件生命周期划分为系统需求分析、软件需求分析、初

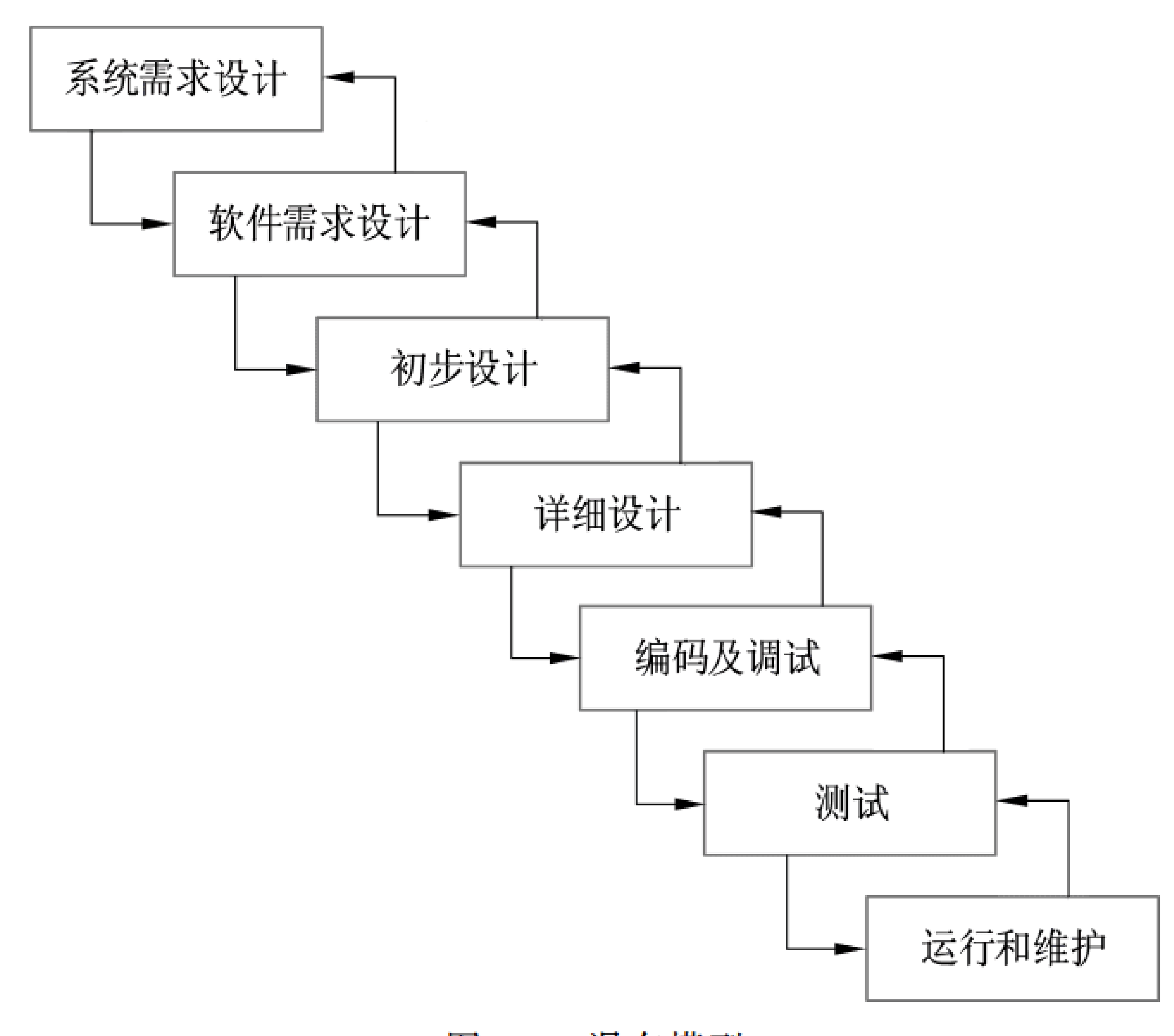


图 1-3　瀑布模型

步设计、详细设计、编码及调试、测试、运行和维护 7 个基本活动，并且规定了它们自上而下、相互衔接的固定顺序，形如瀑布，逐级下落。

瀑布模型是在考虑返回先前阶段纠正系统错误的必要性的情况下，建立软件开发过程模型的第一次全面的尝试。然而这个模型的显著缺点是：它只允许开发人员后退一个阶段。瀑布模型并没有对开发过程后期发现错误时的处理方法作出相应的规定。

2. 迭代模型

迭代模型是统一软件开发过程(Rational Unified Process，RUP)推荐的周期模型，包括产生产品发布(稳定、可执行的产品版本)的全部开发活动和要使用该产品发布必需的所有其他外围元素。开发迭代是一次完整地经过所有工作流程的过程，包括需求工作流程、分析设计工作流程、实施工作流程和测试工作流程。

迭代模型类似小型的瀑布项目，但相对于传统的瀑布模型，迭代过程降低了在一个增量上的开支风险。如果开发人员重复某个迭代，那么损失只是这个开发有误的迭代的花费。另外，由于用户的需求在一开始无法作出完全的界定，而是在后续阶段中不断细化的，因此迭代模型更能适应需求的变化。

每一次迭代都会产生一个可以发布的产品，这个产品将是最终产品的一个子集。迭代模型如图 1-4 所示。

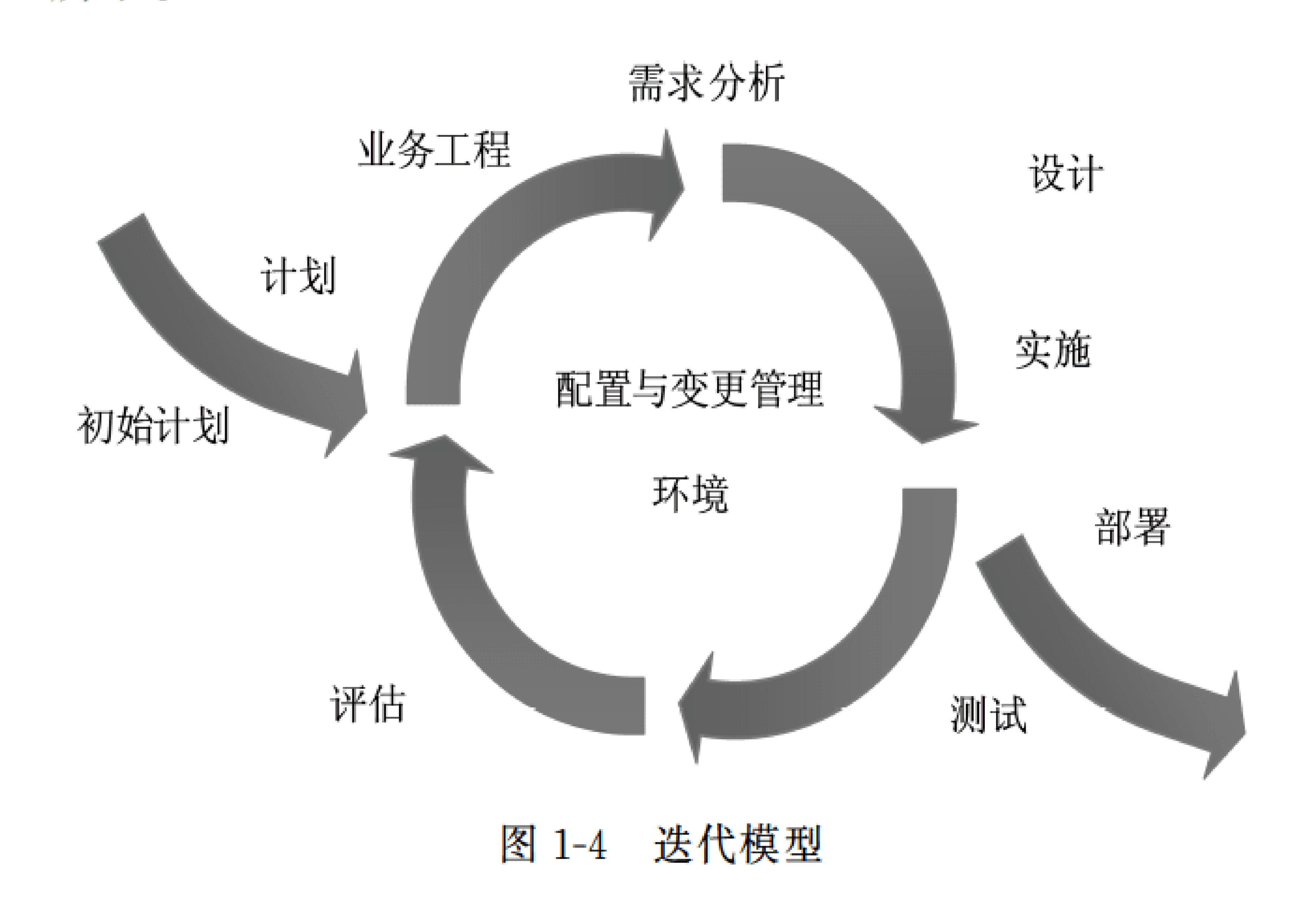


图 1-4　迭代模型

3. 增量模型

增量模型融合了瀑布模型和迭代模型的特征，该模型采用随着日程时间的进展而交错的线性序列，每一个线性序列产生软件的一个可发布的增量。当使用增量模型时，第一个增量往往是核心的产品，即第一个增量实现了基本的需求，但很多补充的特征还没有发布。客户对每一个增量的使用和评估都作为下一个增量发布的新特征和功能，这个过程在每一个增量发布后不断重复，直到产生了最终的完善产品。增量模型如图 1-5 所示。

增量模型与原型模型和其他演化方法一样，在本质上是迭代的，但是与原型模型不一样的是其强调每一个增量均发布一个可操作产品。早期的增量不一定在最终产品版本中体现，但提供了为用户服务的功能，并且为用户提供了评估的平台。

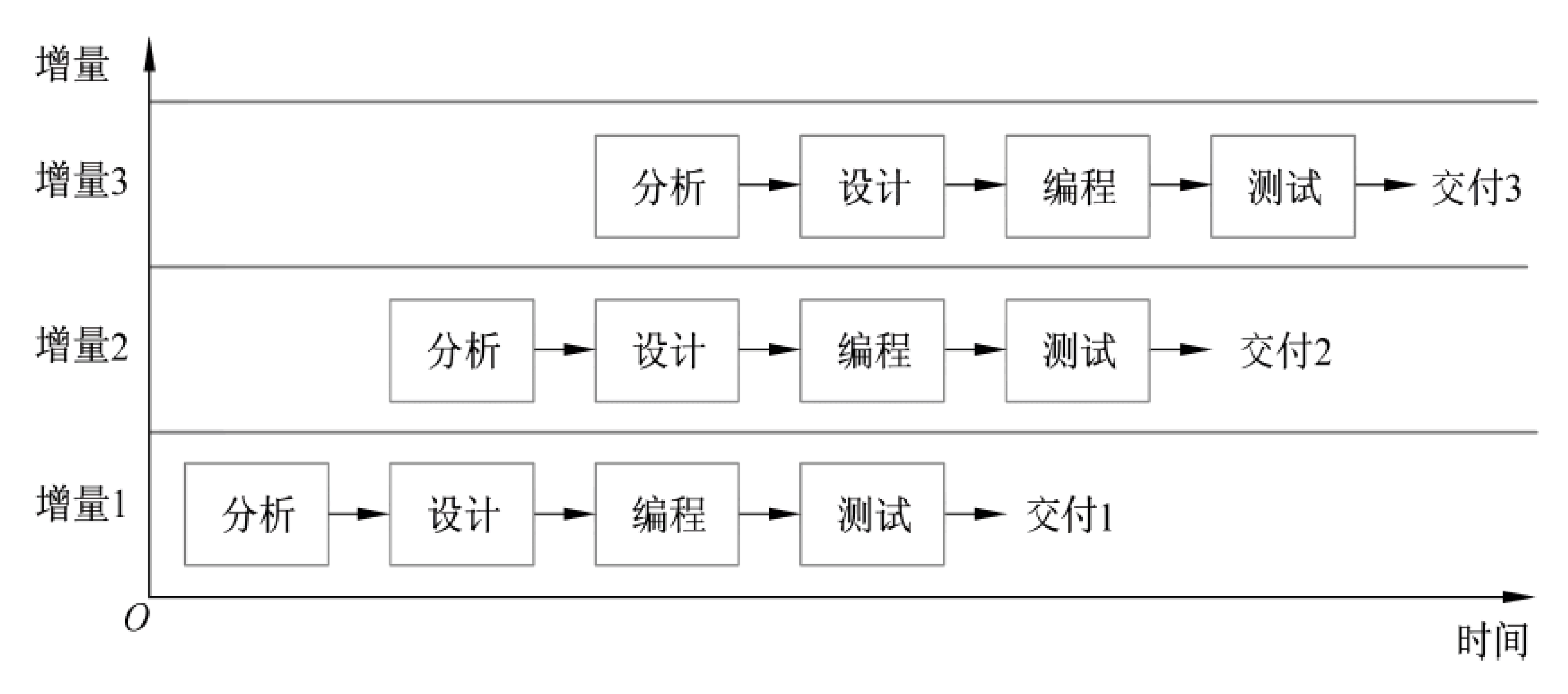


图 1-5　增量模型

4. 快速原型模型

快速原型模型又称原型模型，它是增量模型的另一种形式。快速原型模型是在开发真实系统之前构造一个原型，在该原型的基础上逐渐完成整个系统的开发工作。例如，客户需要一个 ATM 软件，可以先设计一个仅包含刷卡、密码检测、数据输入和账单打印的原型软件提供给客户，此时还不包括网络处理与数据库存取以及数据应急、故障处理等服务。采用快速原型模型进行开发时，第一步是设计一个快速原型，实现客户与系统的交互，客户对这个快速原型进行评价，进一步细化待开发软件的需求。通过逐步调整原型，使其满足客户的要求，开发人员可以确定客户的真正需求是什么。第二步则在第一步的基础上开发客户满意的软件产品。

快速原型是利用原型来辅助软件开发的一种新思想。经过简单快速分析，快速实现一个原型，客户与开发者在使用原型过程中加强沟通与反馈，通过反复评价和改进原型，减少误解，弥补漏洞，适应变化，最终提高软件质量。

5. 螺旋模型

螺旋模型是一种演化软件开发过程模型，它兼顾了快速原型的迭代特征以及瀑布模型的系统化与严格监控特征。螺旋模型最大的特点在于引入了其他模型不具备的风险分析，使软件在无法排除重大风险时有机会停止，以减小损失。同时，在每个迭代阶段构建原型是螺旋模型用以减小风险的途径。螺旋模型更适合大型的、昂贵的系统级软件开发。螺旋模型如图 1-6 所示。

6. 净室模型

净室是一种应用数学与统计学理论，以经济的方式生产高质量软件的工程技术。它力图通过严格的工程化的软件过程实现开发中的零缺陷或接近零缺陷。“净室”一词源自半导体工业中硬件生产车间，通过严格、洁净的生产过程可以预防缺陷的产生，而不是在事后再去排除故障。这个词充分显示了净室模型“防患于未然”的主导思想。

各软件生命周期模型的技术特点和适用范围如表 1-2 所示。

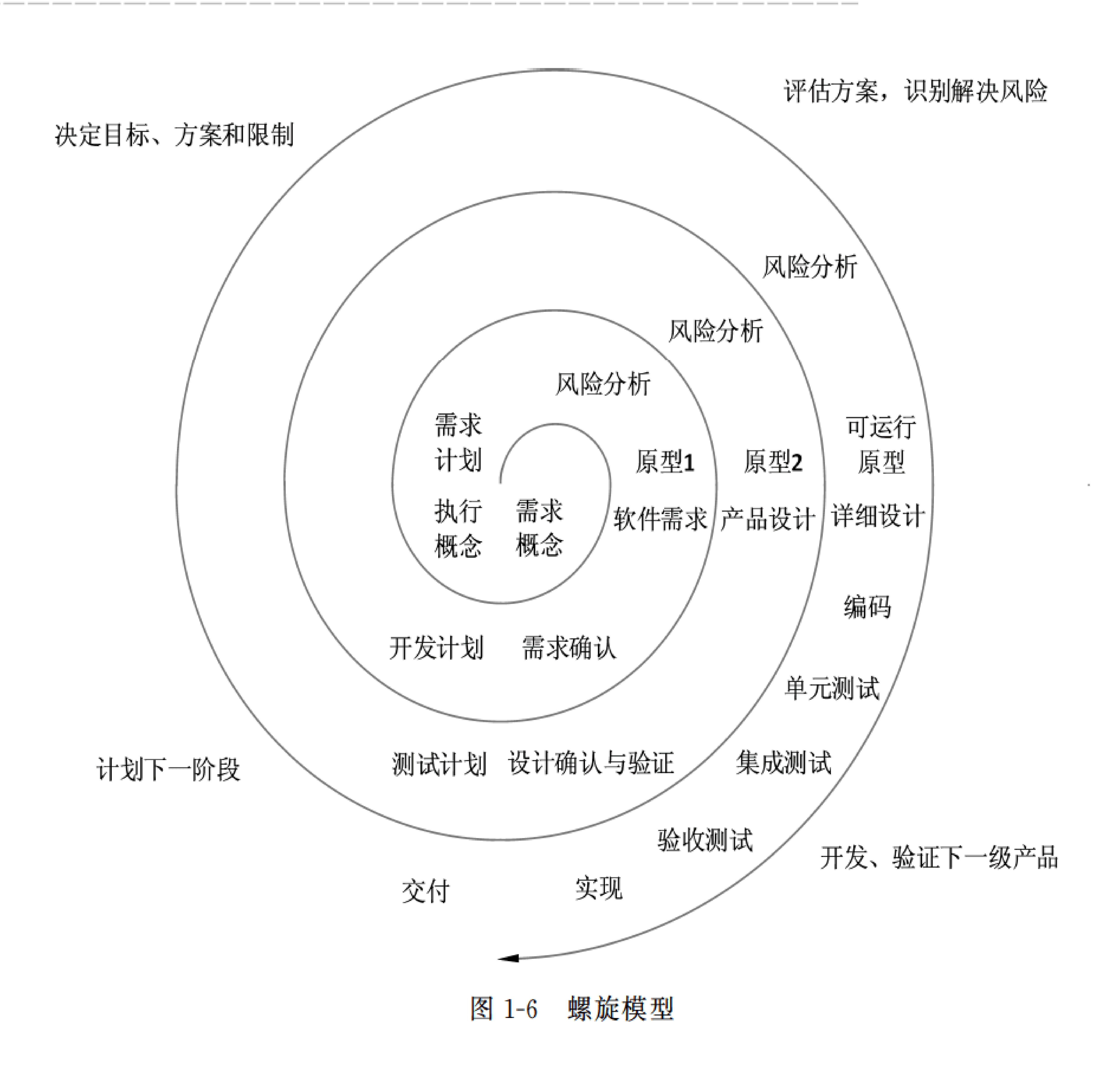


图 1-6　螺旋模型

表 1-2　各软件生命周期模型的技术特点和适用范围

模型名称	技 术 特 点	适 用 范 围
瀑布模型	简单，分阶段，阶段间存在因果关系，各个阶段完成后都有评审，允许反馈，不支持用户参与，要求预先确定需求	需求易于完善定义且不易变更的软件系统
迭代模型	不要求一次性地开发出完整的软件系统，将软件开发视为一个逐步获取用户需求、完善软件产品的过程	需求难以确定、不断变更的软件系统
增量模型	软件产品被增量式地一块块开发出来，允许开发活动并行和重叠	技术风险较大、用户需求较为稳定的软件系统
快速原型模型	不要求预先完备定义需求，支持用户参与，支持需求的渐进式完善和确认，能够适应用户需求的变化	需求复杂、难以确定、动态变化的软件系统
螺旋模型	结合瀑布模型、快速原型模型和迭代模型的思想，并引进了风险分析活动	需求难以获取和确定、软件开发风险较大的软件系统
净室模型	净室技术是一种开发高质量和高可靠性软件的方法，由三大关键技术来刻画：统计过程控制下的增量开发，基于函数的规范、设计和验证，以及统计测试和认证	质量要求非常高的项目；软件开发团队需要具备丰富的数学知识以及与编程语言和开发环境相关的知识和经验

1.3　软件安全开发模型

随着信息技术的高速发展和企业信息化步伐的深入，各单位、企业在利用信息技术实现其目标的同时也变得越来越依赖于信息技术。以银行业为例，为了降低运营成本，提高用户满意度，银行普遍利用信息技术支持的电话银行、ATM 及网络银行系统为客户提供个性化、差异化的服务。另外，银行也在不断利用信息技术收集和分析大量的与客户有关的数据和信息，为新产品开发提供决策支持，从而达到提高盈利水平的目标。在现在的企业信息环境里，如何保证信息系统以及信息本身的安全性已变得至关重要。在今天，具有一定规模的企业的 IT 环境里往往存在着少则几百个、多则上千个大大小小的业务应用，小到支持几个人的小规模团队的协同应用，大到贯穿于一个企业生产、经营、管理全过程的企业级应用。如何降低信息技术给企业带来的各种安全风险已成为当今所有企业面临的新挑战。

使用软件安全开发模型是降低企业信息安全风险的一种有效方法。它从软件生命周期的角度，对安全设计原则、安全开发方法、最佳实践和安全专家经验等进行总结，通过各种安全活动来保证尽可能得到安全的软件。近年来，众多软件安全开发方法和模型涌现出来，比较知名的有微软公司的安全开发生命周期、内建安全成熟度模型、软件保证成熟度模型、综合的轻量应用安全过程等，下面将依次进行介绍。

由于微软公司的安全开发生命周期模型在软件的需求、设计、开发、测试、发布、部署等阶段均为开发人员提供了有价值的安全指导，在安全管控策略、角色分工、安全设计与部署标准等方面均发展得十分成熟，因而本节也按照 SDL 划分的软件开发阶段进行讲述。

1.3.1　安全开发生命周期

安全开发的涵盖面非常广，涉及人员、流程和技术等诸多方面的因素。虽然近些年信息安全受到的关注越来越多，企业在这方面的投入也不断增加，然而各种安全事件却仍然层出不穷。单纯的技术手段已经无法帮助企业彻底消除存在的诸多安全问题，只有建立起一整套安全策略和流程，通过合理地配置资源，充分利用各种技术和非技术手段，企业才有可能更加有效地管理各类安全风险。

安全开发生命周期（Security Development Lifecycle，SDL）是微软公司提出的从安全角度指导软件开发过程的管理模式。SDL 基于 3 个核心概念：教育、持续过程改进和责任。通过对软件开发小组中的技术工作角色进行持续不断的教育，定期评估 SDL 过程，并随着新技术的发展和新威胁引入应对措施，从安全漏洞产生的根源上解决问题，保证产品的安全性。保证和提高应用安全性的最佳时机是在应用的开发阶段，SDL 在开发的所有阶段都引入了安全和隐私的原则。自 2004 年起，SDL 一直是微软在全公司实施的强制性策略。

SDL 将软件开发生命周期划分为 7 个阶段，并提出了 17 项重要的安全活动，如图 1-7

所示。



图 1-7　SDL 的阶段和活动

在这 7 个阶段中，SDL 要求前 6 个阶段的 16 项安全活动为开发团队必须成功完成的必需安全活动，这些必需活动由安全和隐私专家确认有效，并且会作为严格的年度评估过程的一部分，不断进行有效性评析。同时，SDL 认为开发团队应保持灵活性，以便根据需要，选择可选安全活动，如人工代码评析、渗透测试、相似应用程序的漏洞分析，以确保对某些软件组件进行更高级别的安全分析。

下面介绍这 7 个阶段的主要含义和目的。

1. 培训阶段

培训阶段有一项安全活动，即核心安全培训。

开发团队的所有成员都必须进行安全意识与能力的培训，了解相关的安全知识，以确保 SDL 能有效实施，同时能针对新的安全问题与形式持续提升团队的能力。培训对象包括开发人员、测试人员、项目经理、产品经理等。

2. 要求阶段

要求阶段有以下 3 项安全活动：

（1）确定安全要求。在项目确立之前，需要提前与项目经理进行沟通，确定安全要求和需要做的事情。确认项目计划和里程碑，尽量避免因为安全问题而导致项目延期发布。

（2）创建质量门/Bug 栏。质量门和 Bug 栏用于确定安全和隐私质量的最低可接受级别。Bug 栏是应用于整个开发项目的质量门，用于定义安全漏洞的严重性阈值。例如，应用程序在发布时不得包含具有“关键”或“重要”评级的已知漏洞。Bug 栏一经设定，便绝不能放松。

（3）安全和隐私风险评估。安全风险评估和隐私风险评估必须包括以下信息：

- 项目的哪些部分在发布前需要建立威胁模型？
- 项目的哪些部分在发布前需要进行安全设计评析？
- 项目的哪些部分需要不属于项目团队且双方认可的小组（团队）进行渗透测试？
- 是否存在安全顾问认为有必要增加的测试或分析？
- 模糊测试要求的具体范围是什么？
- 隐私内容对评级有哪些影响？

3. 设计阶段

设计阶段通过分析攻击面，设计相应的功能和策略，降低和减少不必要的安全风险。同时通过威胁建模，分析软件或系统的安全威胁，提出缓解措施。

该阶段包括以下 3 项安全活动：

（1）确定安全要求。在设计阶段应仔细考虑安全和隐私问题，在项目初期确定安全需求，尽可能避免由于安全问题引起的需求变更。

（2）减小攻击面。该任务与威胁建模紧密相关，不过它解决安全问题的角度稍有不同。减小攻击面通过减小攻击者利用潜在弱点或漏洞的机会来降低风险，具体包括以下措施：关闭或限制对系统服务的访问，应用“最小权限原则”，以及尽可能进行分层防御。

（3）威胁建模。为项目或产品面临的威胁建立模型，明确攻击可能来自哪些方面。

4. 实施阶段

在该阶段，开发人员按设计要求实现相应的功能和策略，同时通过安全编码和禁用不安全 API 来减少实施时导致的安全问题，不引入编码级安全漏洞，并通过代码审计等措施确保安全编码规范的实行。

该阶段包括以下 3 项安全活动：

（1）使用批准的工具。开发团队使用的编辑器、链接器等相关工具可能会涉及一些与安全相关的环节，因此在使用工具的版本上需要提前与安全团队进行沟通。

（2）弃用不安全的函数。许多常用函数可能存在安全隐患，应当禁用不安全的函数和 API，使用安全团队推荐的函数。

（3）静态分析。代码静态分析可以由相关工具辅助完成，并与人工分析相结合。

5. 验证阶段

在该阶段，通过安全测试的手段检测软件的安全漏洞，并全面核查攻击面，确认各个关键因素上的威胁缓解措施是否得以正确实现。

该阶段包括以下 3 项安全活动：

（1）动态分析。动态分析是静态分析的补充，用于在测试环节验证程序的安全性。

（2）模糊测试。这是一种专门形式的动态分析，它通过故意向应用程序引入不良格式或随机数据来诱发程序故障。模糊测试策略的制定以应用程序的预期用途、功能和设计规范为基础。安全顾问可能要求进行额外的模糊测试，或者扩大模糊测试的范围和增加持续时间。

（3）威胁模型和攻击面评析。项目经常会因为需求等因素导致最终的产出偏离原本设定的目标，因此在项目后期对威胁模型和攻击面进行评析是有必要的，能够及时发现问题并加以修正。

6. 发布阶段

在该阶段，建立事件响应计划，进行最终安全评析，并完善相应的安全指导文档以提交给用户使用。

该阶段包括以下 3 项安全活动：

(1) 事件响应计划。受 SDL 要求约束的每个软件在发布时都必须包含事件响应计划。即使在发布时不包含任何已知漏洞的产品,也可能在日后遇到新出现的威胁。需要注意的是,如果产品中包含第三方的代码,也需要留下第三方的联系方式并加入事件响应计划,以便在出现问题时能够找到对应的人。

(2) 最终安全评析。在发布之前仔细检查对软件执行的所有安全活动。

(3) 发布/存档。在通过最终安全评析后,或者虽有问题,但达成一致后,可以完成产品的发布。发布的同时需要对各种问题和文档进行存档,为紧急响应和产品升级提供帮助。

7. 响应阶段

在该阶段,响应安全事件与漏洞报告,实施漏洞修复和应急响应。同时发现新的问题与安全问题模式,学习新的知识,将它们用于 SDL 的持续改进过程中。

该阶段有一项安全活动,即执行事件响应计划。

1.3.2 内建安全成熟度模型

内建安全成熟度模型(Building Security In Maturity Model,BSIMM)是通过对多个软件公司的软件安全项目进行研究而得到的模型,该模型对不同软件安全项目所采取的措施和实践进行量化,描述其共性和各自的特点。

很多软件公司一方面希望能构建功能强大而又安全的软件,另一方面却不知道该如何构建,没有其他公司的最佳实践作为参考。针对此问题,BSIMM 通过分析多个真实的软件安全项目的实践数据,提炼出这些软件公司和软件项目中的共同特点,帮助更多的软件公司对其项目进行安全规划、实现和评测。因为 BSIMM 中分析的原始数据来自当前国际上很多著名的大型公司,包括美国银行(Bank of America)、美国第一资本银行(Capital One)、微软(Microsoft)、谷歌(Google)、英特尔(Intel)以及赛门铁克(Symantec)等,所以其分析结果体现了当今业界软件安全技术的领先水平,具有一定的可信赖度。

例如,BSIMM 5 的软件安全框架(software security framework)描述了软件安全开发的 4 个领域、12 项实践,具体如图 1-8 所示。

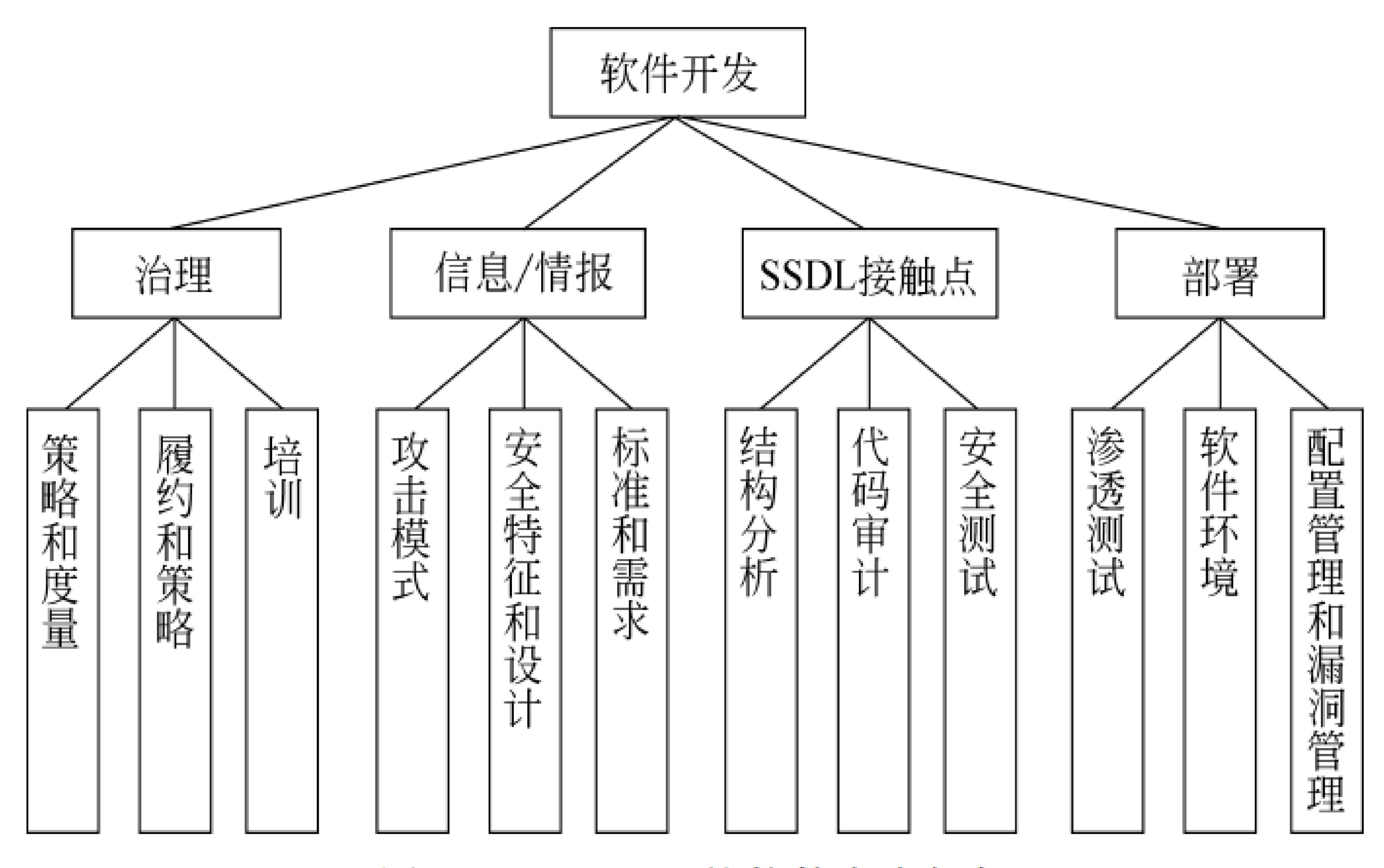


图 1-8 BSIMM 的软件安全框架

（1）治理（governance）。帮助组织、管理和衡量软件安全措施，员工的发展培训也是治理的核心实践之一。治理领域包括策略和度量、履约和策略以及培训 3 项基本实践。

（2）信息/情报（intelligence）。收集使软件安全措施应用于整个组织的知识，收集内容包括主动安全指导和组织威胁建模。信息/情报领域包括攻击模式、安全特征和设计以及标准和需求 3 项基本实践。

（3）SSDL 接触点（SSDL touchpoint）。SSDL 意为软件安全开发生命周期。SSDL 接触点领域分析和保障特定软件开发工具和过程，包括各种软件安全方法。SSDL 接触点领域包括结构分析、代码审计与安全测试 3 项基本实践。

（4）部署（deployment）。解决软件配置、维护以及其他对软件的安全性有直接影响的环境问题。部署领域包括渗透测试、软件环境、配置管理和漏洞管理 3 项基本实践。

1.3.3 软件保证成熟度模型

软件保证成熟度模型（Software Assurance Maturity Model，SAMM）提供了一个开放的框架，用以帮助组织和企业制定并实施有关软件安全的特定风险策略。SAMM 于 2009 年正式发布 1.0 版本，目前由 OWASP（Open Web Application Security Project，开放 Web 应用安全项目）组织作为一个开放的项目来维护。SDL 一般适用于软件开发商，它们以开发软件为主要业务，其软件开发技术较成熟，有严格的质量控制。而与 SDL 相比，SAMM 更适用于自主研发软件的组织，如银行、在线服务提供商等，它们更强调效率。

SAMM 提供的资源可用于以下目的：

- 评估一个组织已有的软件安全实践。
- 在明确定义的迭代中建立平衡的软件安全方案。
- 证明安全保证计划带来的实质性改善。
- 定义并衡量组织中与安全相关的措施。

SAMM 规定了软件开发过程中的 4 个核心业务功能，包括治理、构造、确认以及部署，这 4 个业务功能各包括 3 项安全实践。SAMM 的每个安全实践是一个与安全相关的措施，以便保证相关业务功能的实现。所以，从总体来说，这 12 项安全实践都是改进软件开发业务功能的独立部分。具体如下：

（1）治理。专注于软件开发企业组织管理其软件安全开发相关的过程、活动和措施，主要包括战略与度量、策略与遵循、培训与指导 3 项安全实践。

（2）构造。关注软件安全开发中需求、目标和架构方面的过程、活动和措施，主要包括威胁评估、安全需求和安全架构 3 项安全实践。

（3）确认。注重软件检查和测试中的过程、活动和措施，主要包括设计审查、代码审查和安全测试 3 项安全实践。

（4）部署。强调软件发布和部署配置时相关的过程、活动和措施，主要包括漏洞管理、环境加固和操作激活 3 项安全实践。

SAMM 的总体框架如图 1-9 所示。

对于每一项安全实践，SAMM 设置了 3 个成熟度等级和 1 个隐含的零起点。它们的细节对不同实践有所不同，但它们普遍代表的含义如下：

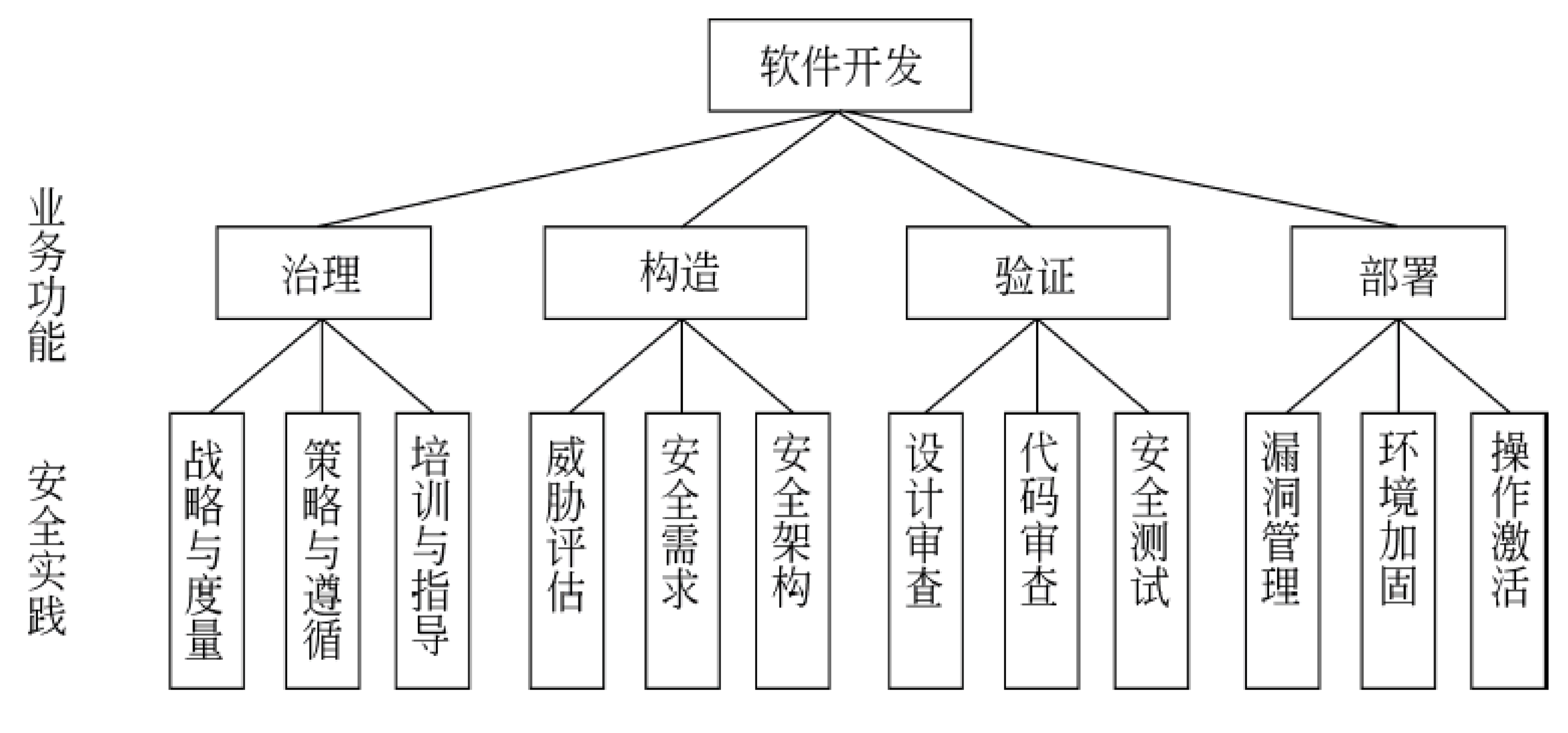


图 1-9　SAMM 的总体框架

0：隐含的 0 起点，代表实际没有实现该安全实践。

1：对安全实践有了初步了解并有专门的设计和运用。

2：进一步提高了安全实践的效率和有效性。

3：在一定规模上综合、有效地掌握并运用了安全实践。

对于某个特定的软件开发项目或软件开发企业，使用 SAMM 对其进行评估时有两种推荐的评估方法：

(1) 简单方法。对每一项安全实践进行评估，并为得到的评估结果评定分数。

(2) 详细方法。对每一项安全实践进行评估后，再执行额外的审计工作，以确保每个安全实践中规定的每一项措施都已执行，且已达到成功指标。

软件开发企业可以以 SAMM 为基准来衡量其软件安全保证计划。通过使用评估和记分卡，软件开发企业能够证明其软件安全性得到循序渐进的改善。软件开发企业还可以参考 SAMM 路线图模板，以建立或改善一个软件安全保证计划。

1.3.4　综合的轻量应用安全过程

综合的轻量应用安全过程(Comprehensive Lightweight Application Security Process，CLASP)最初由安全软件公司(Secure Software，Inc.)提出，后来由 OWASP 完善、维护并推广。

CLASP 是一个用于构建安全软件的轻量过程，包括由 30 个特定的活动和辅助资源构成的集合，用于提升整个开发团队的安全意识，并针对这些活动给出了相应的指南、导则和检查列表。它使企业能够使用简便、快捷而系统的流程开发出安全的软件产品，并且也能够和多种软件开发模型相结合。

CLASP 的主要特点是其安全活动基于角色安排。它强调安全开发过程中的角色和职责，软件开发不同阶段的安全活动需要指派不同的角色负责和参与。这些角色包括项目经理(Project Managers)、需求专家(Requirements Specifiers)、软件架构师(Software Architects)、设计者(Designers)、实施人员(Implementers)、集成和编译人员(Integrator and Assemblers)、测试者和测试分析师(Testers and Test Analysts)、安全审计员(Security Auditors)。对于每一项基于角色的安全活动，CLASP 都对以下问题进行了描述：

• 安全活动应该在什么时间以及如何实施?
• 如果不进行这项安全活动,将会带来多大的风险?
• 如果实施这项安全活动,估计需要多少成本?

CLASP 的主要目标是支持构建安全处于中心地位的软件,其安全活动大都从安全理论的角度定义和构思,因此,安全活动的覆盖面相当广泛。同时,CLASP 通过一些安全最佳实践将安全属性以一种结构化、可重复和可测量的方式整合进软件开发企业的现有或者将要展开的软件开发生命周期中。其相互独立的安全活动以结构化组织的方式集成到开发过程和运行环境中。为了保持灵活性,对要执行的安全活动及其执行顺序的选择是开放的,不同开发者可以自行裁剪以适应待开发产品的实际情况。

1.4 人员角色规划

随着软件规模的不断膨胀,软件开发人员的组织和分工也变得越来越复杂,对软件开发人员如何合理地进行组织和分工越来越成为一个产品能否成功开发的决定性因素。人员角色规划主要从人员角色设置、安全开发培训、安全开发过程管理等方面将 SDL 运用于软件开发生命周期,提高开发人员的安全意识,整合安全开发流程,提升软件产品质量。

对于软件安全开发来说,参与人员角色通常有以下几种:

(1) 管理层。企业安全策略的推进需要管理层的大力支持。为了更好地进行软件安全开发,负责推进软件安全开发的主管领导应由信息部门的管理层兼任,以利于流程的真正落地。

(2) 软件安全团队。这是保证软件安全开发的基础,在启动软件开发生命周期各阶段安全活动之前应组建一个软件安全团队,否则无法成功执行各项安全活动。软件安全团队负责解决下列问题:安全活动应该在什么时间实施以及如何实施?如果不进行这项安全活动,将会带来多大的风险?如果实施这项安全活动,估计需要多少成本?

(3) 业务人员。该角色在传统需求分析的基础上融入安全性考量,进行安全需求分析。

(4) 架构师及开发人员。架构师需要在整个过程中对技术活动和工件进行领导和协调,确立架构图的整体结构。开发人员在进行开发前应接受安全培训,在开发过程中应遵循安全编码规范,安全开发,从根本上减少软件安全问题。

(5) 测试人员。负责使用相关工具或方法对系统进行功能测试、性能测试和安全测试,其中包括设置和执行测试、评估测试过程并修改错误,以及评估测试结果并记录所发现的缺陷等工作。

(6) 其他人员。包括厂商、运维人员等。对这些人员也需进行安全开发培训,以增强其软件安全开发意识。

第 2 章 软件安全需求与设计

同传统的软件开发过程一样，安全需求分析在软件安全开发过程中也是必不可少的一个过程。安全需求分析阶段的一个很小的改进可能会带来很高的回报率。在软件需求定义和分析完成后，即进入软件设计阶段，设计的好坏将直接影响软件产品的质量和用户对最终软件产品的满意程度。本章首先介绍安全需求的定义和分析方法，其次介绍设计阶段安全设计的原则和方法，最后介绍威胁建模。

2.1 安全需求概述

2.1.1 安全需求的定义

随着新技术的应用与发展，现在的软件系统越来越复杂，用户对软件的功能、性能以及安全性的要求越来越高，仅通过传统的需求分析方法来了解、描述软件性能已远远不够，在需求分析阶段融入安全需求分析方法更能满足用户真实的安全需求，并可以减少后期维护带来的巨额花销。

一般最终的软件需求规格说明是由功能性需求、非功能性需求和约束构成的。在安全需求的定义中，最广泛的一种观点是将安全需求看作非功能性需求，是产品功能描述的补充，从安全方面描述产品的各种特性；有的观点将安全需求看作与业务功能需求同等重要的安全功能需求，它是开发人员必须实现的软件功能，并对业务功能需求具有约束力；还有的观点使用安全策略或质量需求来定义安全需求。当开发一个新的安全信息系统时，开发人员在安全需求分析阶段需要解决以下问题：系统的安全目标是什么，系统应该提供什么样的安全服务，工作将受到什么条件约束，等等。

软件安全需求是为保障实现业务功能而对相关信息的机密性、完整性和可用性提出的要求，一般通过 3 个属性来描述对于数据和信息计算服务的基本安全目标：

(1) 可信性(confidentiality)，指保护敏感信息不被未授权用户访问。

(2) 完整性(integrity)，指保护数据不被更改或破坏。

(3) 可用性(availability)，指确保资源被授权的用户使用。

在定义安全需求时，应在安全系统的开发人员和提出需求的用户之间建立起一种理解和沟通的机制，以一种清晰、简洁、一致且无二义性的方式，对一个待开发的安全系统中各个有意义的方面进行陈述。安全需求分析中应包含足够多的信息，以使开发人员能够编写一个满足用户真实的安全需求的系统。

安全需求分析应遵循以下原则：

(1) 依据标准。为了保证需求分析的质量，安全人员需要充分参考、利用现有的技术标准来衡量和判断系统有哪些安全要求。

(2) 分层分析。安全需求分析涉及策略、体系结构、技术、管理等各个层次的工作，通过分层分析得出的结果更加完备、可靠。

(3) 结合实际。安全需求分析需要针对具体的信息系统环境进行，所有工作的开展必须建立在实际环境基础之上。在不同环境中，安全需求分析的结果是不同的。

2.1.2　安全需求的标准

系统对安全方面的需求有很多，涉及待开发系统的功能、性能和约束等各个方面的内容。为了更好地规范安全需求，国际组织和各国制定了一系列常用的安全标准供开发人员参考，例如《信息技术 安全技术 应用安全》(ISO/IEC 27034)、《信息技术 软件安全保障规范》(GB/T 30998—2014)、功能安全基础标准 IEC 61508、信息技术安全性认证通用标准(CC 标准)等，本节就 CC 展开描述。

《信息技术安全评估通用标准》(Common Criteria for Information Technology Security Evaluation，简称 CC)的原则是基于对系统安全可信度的评估提供安全保障。CC 将安全需求划分为安全功能需求(security functional requirements)和安全保障需求(security assurance requirements)两个独立的范畴来定义。

安全功能需求描述的是安全系统应该提供的安全功能。开发人员在确定系统的安全需求和安全目标时，可以参照 CC 第二部分“安全功能要求”中已定义的安全审计(FAU)、通信/不可抵赖(FCO)、密码支持(FCS)、用户数据保护(FDP)、标识和鉴别(FIA)、安全管理(FMT)等，根据不同情况裁剪开发系统所需具备的安全功能，开发人员通过参考 CC 的安全功能需求类别可以有效防止开发完成的系统存在安全组件缺失和遗漏的情况。

安全保障需求描述的是安全可信度及为获取一定的可信度而应该采取的措施。CC 第三部分“安全保障要求”中定义了 10 个类别的安全保障要求，包括保护轮廓评估(APE)、安全目标评估(ASE)、配置管理(ACM)、交付和运行(ADO)、开发(ADV)、指导性文档(AGD)、生命周期支持(ALC)、测试(ATE)、脆弱性评估(AVA)和保障维护(AMA)。这 10 个安全保障需求类别有助于开发人员提高开发系统的安全保障能力，减少系统出现脆弱性的可能性，降低有意利用或无意触发脆弱性的可能性以及由被触发的脆弱性导致的破坏程度。

此外，CC 给出了一套评价系统安全可信度的指标——安全保障级别(EVL)。EVL 通过安全系统在构造管理、发行与操作、开发、指南文档、生命周期支持、测试和脆弱性评估等方面所采取的措施来确定系统的安全可信度。EVL1～EAL7 对开发系统的安全要求不断增加，通过确定需要的 EAL 保障级别，在开发中参考使用并在开发结束后展开相应测评，可极大地提高软件的安全性。

除了 EAL 这一定性的评价指标外，可以用于评价系统安全保障程度的指标还有很多，尤其是一些定量指标(例如系统的病毒感染率和平均失效时间等)，对用户来说更加直观，更加易于理解，在现有的评价安全保障程度的定量指标中，安全风险值是当前较为流

行，也是比较成熟的一种指标。CC中既没有排斥也没有评价这些指标，待这些定量指标的计算和评价方法进一步成熟之后，用户对这些定量指标的要求也将被纳入对系统的安全保障需求中。

2.2 安全需求分析方法

2.2.1 安全需求分析过程

安全需求分析是一个不断发展的过程，不会有一劳永逸的分析结果。由于数据的不确定性和环境的不断变化，要想保持分析结果的有效性，对安全需求的内容需要不断地进行修正，以保证结果时刻最新。在早期的开发实践中将需求工程作为项目开发初期的活动。随着近年来需求分析在软件开发过程中重要性的提高，需求分析已经成为一个持续的过程，跨越整个项目的生存周期，以最终指导系统安全保障措施的实施。

总体来说，软件安全需求的分析过程有以下几个基本步骤。

1. 系统调查

了解系统所处的安全环境（存在于系统边界之外并对系统的安全具有潜在的或直接影响的所有因素）及其他与安全相关的信息，例如用户、组成部件、运行机制及与其他系统的连接情况等。在此基础上确定需要保护的资产，其中可能包括硬件、软件、数据、文档和计算机服务等，并评价各个资产的相对价值。

2. 分析系统的脆弱点和安全威胁

这一步的目标是确定系统暴露的各种脆弱点及面临安全威胁的可能性。通过分析系统环境和常见脆弱点确定系统脆弱点。在系统的脆弱点被确定后，开发人员需要针对每个脆弱点分别定性和定量地分析由此可能引发的安全威胁及其对资产可能造成损害的程度。对脆弱点和安全威胁的可能性进行估算是非常困难的，因为这种可能性与当前采用的安全措施和所处的安全环境有关。目前主要采用概率统计的方法，通过分析操作日志、局部入侵的统计和用户的投诉等数据，进一步计算出系统承受的安全风险值。

3. 需求的确定

安全需求分析的最后一个阶段将定性分析和定量分析的结果结合起来以定义信息系统的安全需求，开发人员由此确定需要实施的相应的安全措施，为信息系统提供有效而且合理的安全保障。

2.2.2 安全需求分析的常用方法

目前应用的安全需求分析方法大多是对传统需求分析方法的扩展，即在传统需求分析过程中融入安全性考量。本节主要介绍滥用案例、滥用框架和安全质量需求工程(Security Quality Requirements Engineering，SQUARE)这3种常用的安全需求分析方法。

1. 滥用案例

滥用案例(abuse case)是在传统的需求分析方法用例上的安全扩展,强调了需求分析中的安全性。开发者通过将自己置于攻击者的状态对系统进行系统利用和破坏,考虑当安全机制无效或被破坏时的后果,从而更加深入地了解系统,防范可能发生的恶意攻击。

UML、用例等建模和设计工具可以帮助软件开发人员规范地描述和设计软件的行为,但使用这些建模和设计工具的前提是:软件用户的所有行为都是正确的,这意味着开发人员是基于系统不会被有意滥用的假设来理解系统的完全功能的。而当系统被有意滥用时,其表现结果是未知的。

滥用案例的典型方法有误用用例和滥用用例。误用用例方法是指从功能性用例的文本描述中分析可能存在的安全漏洞并识别出对应的威胁,建立威胁用例,针对威胁用例建立安全需求用例。滥用用例方法主要用于捕获攻击者与系统之间的交互所产生的威胁,该方法重视对攻击者的描述,主要对攻击者的企图、攻击能力进行评估,滥用用例方法针对识别出的威胁,单独建立威胁用例。与误用用例方法不同的是:滥用用例建立好的威胁用例并不与功能性用例产生交互,而是仅说明系统面临的安全威胁。威胁用例的描述形式既可以采用已有的用例模板,也可采用漏洞攻击树。

滥用用例通过下面的 5 个步骤创建:

(1) 用 UML 的方法描述参与者和用例。

(2) 引入主要的滥用者和滥用用例。

(3) 研究滥用用例和用例之间潜在的 include 关系。

(4) 引入新的用例来发现或阻止滥用用例。

(5) 形成详细的需求记录。

2 滥用框架

滥用框架(abuse frame)是在传统的需求分析方法中问题框架上的安全扩展,滥用框架方法是一种面向问题域的分析方法。该方法从攻击者的角度考虑系统面临的问题,采用已有的问题框架方法来支持工具分析和获取安全需求,适用于针对问题领域进行分析,获取安全需求,其目的是在系统发生违反安全行为的条件下,使系统呈现安全威胁,以此对系统进行分析。

滥用框架通过滥用框架威胁图和安全需求描述图描述系统面临的安全问题。滥用框架威胁图中各个领域的现象描述以及领域之间的现象描述可作为安全知识,帮助同领域内的其他软件系统确定安全威胁。定义攻击者领域,表示攻击者;定义受害者领域,表示系统遭受威胁的资产。引入“反需求”表示攻击者对系统的需求,先获取反需求,再获取相应的安全需求。

滥用框架的威胁图如图 2-1 所示,主要从两个方面入手:一是从问题领域内的现象入手,由于现象描述缺乏安全性约束,使得安全漏洞可能显式或隐式地存在于现象中;二是从领域之间的现象交互过程入手,其存在的安全隐患经常是外部攻击的切入点。在图 2-1 中,E1 表示反需求在受害者领域产生的安全威胁现象,E2 表示机器领域和受害者领域之间的共享现象,机器领域通过 E2 对受害者领域进行操作来完成软件系统的功能,

实线的 E3 表示攻击者对机器领域进行攻击时产生的现象，虚线的 E3 表示反需求受到攻击者领域的需求引用，它和反需求一起描述了攻击者领域对机器领域的攻击效果。

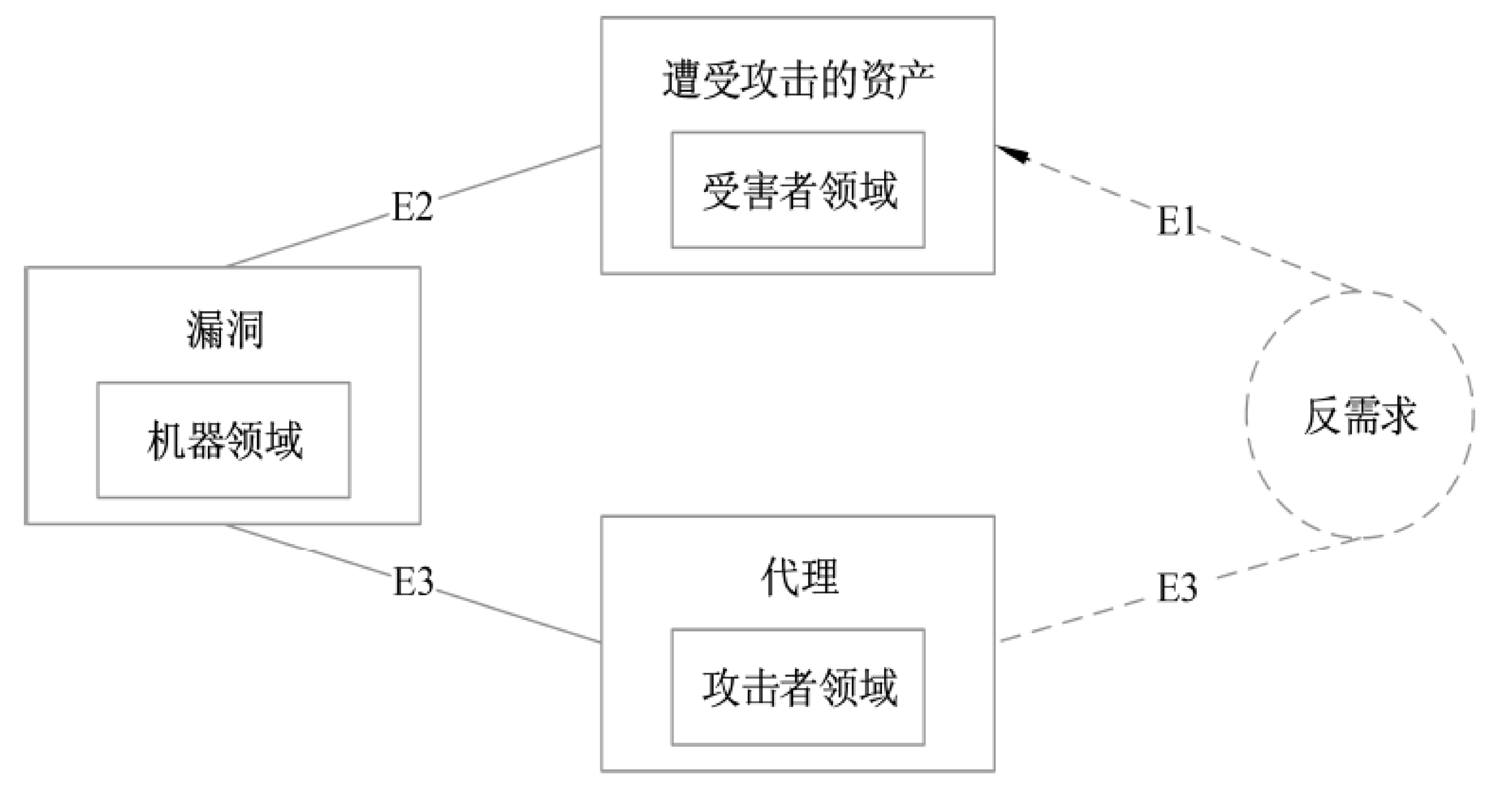


图 2-1　滥用框架威胁图

针对反需求，滥用框架方法使用问题框架来描述安全需求，其安全需求描述图如图 2-2 所示，E1 表示在受到攻击的情况下受保护领域所期望的现象，E2 和 E3 的含义与图 2-1 一致。

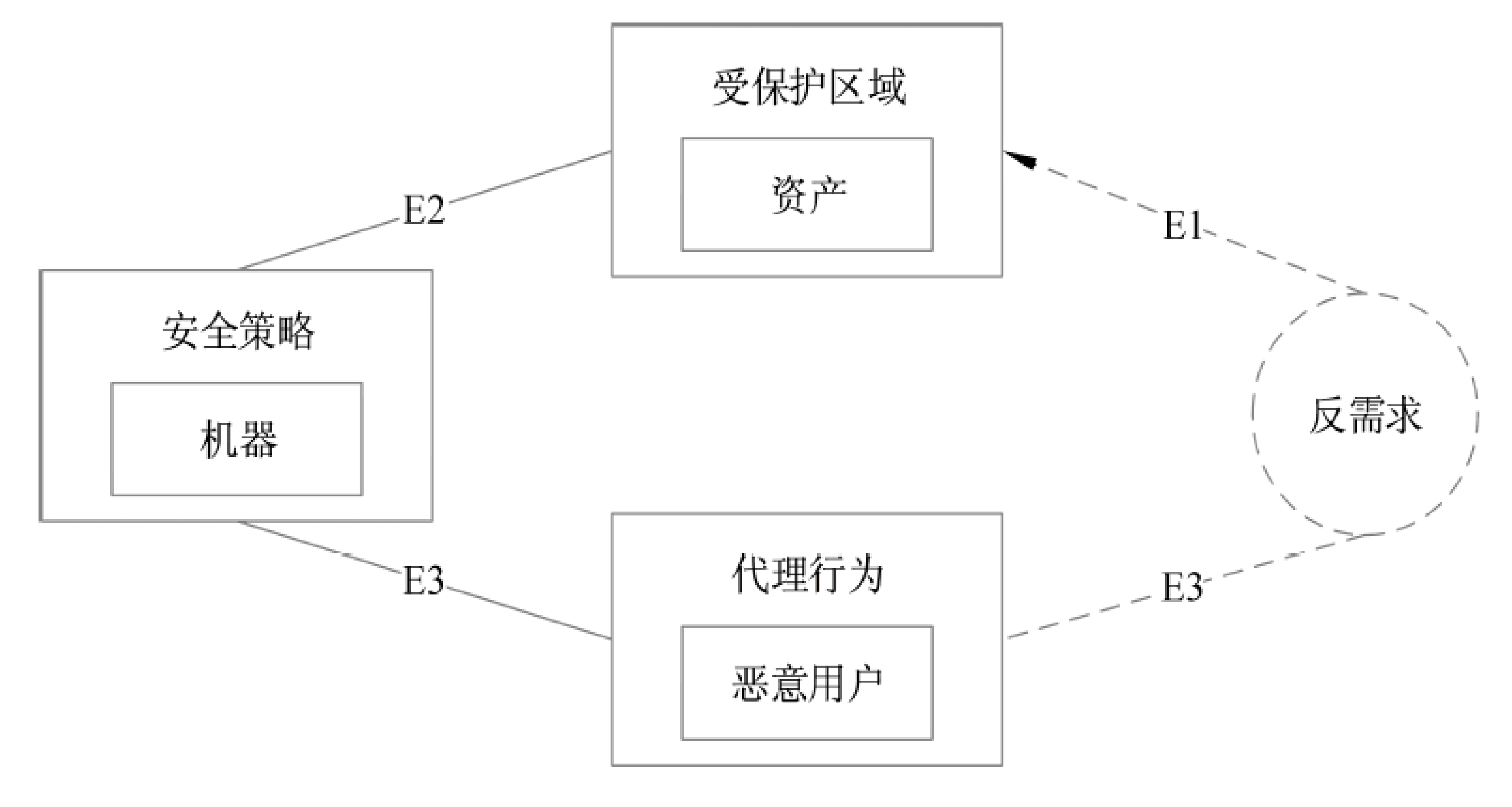


图 2-2　安全需求描述图

3. SQUARE 模型

SQUARE 模型的特性是在软件开发周期的早期植入安全概念，在系统实现后通过此模型分析系统的安全性，并对系统将来的修改和更新起重要作用。使用 SQUARE 模型时，软件项目的安全开发过程必须考虑其运行环境。例如，一个运行在孤立工作站的系统的安全需求和一个运行在基于网络服务器的系统的安全需求有很大区别。这样的需求差异可在 SQUARE 过程中的第三步加以区分，在制订开发方案时考虑到不同的场景。此外，当一个项目发生变化时，应当重新应用 SQUARE 过程分析安全需求。SQUARE 的基本过程如下：

(1) 统一定义。这是安全需求工程的首要条件，开发团队根据经验来定义软件项目的安全内容，这个定义可能与项目的实际安全要求差别很大。通过查阅 IEEE 和

SWEBOK 等安全相关资料，有利于开发团队更加准确地进行软件项目安全内容的统一定义。

(2) 确认安全目标。在软件项目开始阶段就必须确定其安全目标，并且在整个软件项目生命周期中持续关注安全目标。软件项目的不同用户有不同的安全目标。例如，人力资源部的客户只关心人力资源档案的保常性，而财务部的客户只关心财务数据在没有授权的情况下不能轻易访问或修改。

(3) 开发方案。其目标是得到可以更好地支持安全需求定义的方案，它是安全需求工程活动必需的过程。这一过程输出的开发方案资源可能包括误用/滥用案例、模板、框架等。

(4) 进行风险评估。这一活动需要风险专家、投资者和安全需求工程师的支持。风险专家会根据软件项目的需求推荐特定的风险评估方法，进行风险分析并得出风险评估结果，这是得到准确的安全需求的必要步骤。

(5) 选择启发式方法。当软件项目的客户包含不同类型时，这一活动尤其重要。因为选择较为形式化的启发式方法能够很好地克服不同知识背景的客户之间的沟通问题。

(6) 得出安全需求。这一步骤建立在前面所有步骤的基础之上，利用所选方案(如滥用案例)得出安全需求，它是使用一定选择技巧的启发式方法。

(7) 需求分类。安全需求工程师区分关键的需求、目标(理想的需求)和可能存在的构架约束。这个分类有助于后面的需求排序过程。

(8) 需求排序。对需求的排序不仅依赖于需求分类，也关系到成本效益分析，这些分析可以决定哪些安全需求有很高的效益。

(9) 需求检查。检查需求的方法很多，例如专用检查方法和同行审查。此活动完成后，开发团队会得到一系列排好序的初始安全需求。在后续的开发过程中，开发团队需要注意目前还不完善的需求，并理解哪些需求独立于特定的架构和实现。

2.3　系统设计概述

2.3.1　系统设计内容

软件设计是软件工程的核心，是软件开发生命周期最重要的阶段。开发人员依照软件设计来实现系统的功能和性能，如果软件在设计阶段存在安全缺陷，如业务逻辑缺陷，用户将面临危险，损失无法估量，程序员最终还需在解决安全问题上花费大量时间。因此，好的设计是开发出高质量软件的基础。

从生命周期的角度，软件设计可以看作从软件需求规格说明书出发，根据需求分析阶段确定的功能，设计软件系统的整体结构，划分功能模块，确定每个模块的实现算法，结合软件安全技术和安全设计原则产生满足功能需求和非功能需求的具体设计方案，即从整体到局部，从总体设计到详细设计的过程。

1. 总体设计阶段

系统的总体设计又称概要设计，此阶段在较抽象的层次上对比、分析多种可能的系统

实现方案和软件结构，从中选出最佳方案和最合理的软件结构。总体设计阶段需要完成应用系统的安全整体架构设计，包括但不限于安全体系结构设计、各功能块间的处理流程、与其他功能的关系、安全协议设计、安全接口设计等。

软件总体设计阶段的安全设计工作有以下两个：

(1) 系统设计。划分出组成系统的物理元素，如程序、文件、数据库、人工过程和文档等。

(2) 结构设计。将软件需求转化为数据结构和软件的系统结构，综合应用设计、成本、安全性、用户体验等方面选择最佳的系统结构，即确定系统中的每个程序都是由哪些模块组成的，实现需求分析阶段拟定的全部软件安全性需求(包括不希望事件)，确定模块相互间的关系。

2.详细设计阶段

系统的详细设计是在总体设计的基础上进一步细化，在框架中填入细节，得到软件的详细数据结构表达和具体算法描述。详细设计阶段作为安全功能的程序设计阶段，应当直接指导安全功能的编码工作，包括但不限于模块设计、内部处理流程、数据结构、输入输出项、算法、逻辑流程图等。

软件详细设计阶段的安全设计工作有以下4个：

(1) 为每个模块确定采用的算法，选择某种适当的工具表达算法的过程，写出模块的详细过程性描述。

(2) 确定每一个模块使用的数据结构。

(3) 确定模块接口的细节，包括对系统外部的接口和用户界面，与系统内部模块之间的接口，以及模块输入数据、输出数据和局部数据的全部细节。在详细设计结束时，应该把上述结果写入详细设计说明书，并且通过复审形成正式文档，交付下一阶段(编码阶段)作为工作依据。

(4) 为每一个模块设计一组测试用例，以便在编码阶段对模块代码(即程序)进行预定的测试，模块的测试用例是软件测试计划的重要组成部分，通常应包括输入数据、期望输出等内容。

2.3.2 安全设计原则

软件安全设计要求在整个软件生命周期运用系统安全性工程技术确保软件采用提高系统安全性的有效措施，并确保所有可能降低系统安全性的错误都已经被排除或被控制在可接受的风险度以下。为了使软件更好地达到软件安全设计要求，软件设计应该尽量遵循安全设计原则，这些原则是软件开发和软件测试中有关安全经验的高度总结，能够指导安全开发人员(特别是软件架构师和设计师)开发更为安全的软件。下面是一些基本的安全设计原则。

1.简单易懂原则

越复杂越容易出错。越复杂越难审查得透彻，也越难测试得比较全面，从而使得一些安全缺陷遗留在系统中。随着系统复杂度的提高，系统的安全风险也在变高。因此，应减

少代码冗余，提高组件重用度。

2. 最小特权原则

最小特权原则是指：对于请求存储资源的主体，只授予执行操作所需的最少访问权，而且应该保证对于该访问权只准许使用所需的最少时间。如果授予一个用户或进程、组件超过其行为必要的权限范围的许可，该用户或进程、组件就有可能获得或修改其没有权限处理的信息，出现滥用特权的风险。

3. 权限分离原则

尽量把软件划分为多个独立的模块，把权限分离成不同的权限许可和认证条件，把用户分离成不同的权限角色。不要将权限一次性授予一个用户，而是根据需要提供多重的认证与检查机制，再进行授权。权限分离，出现问题时可以快速定位到模块，以便进行修复。还可以对单个模块进行测试，保证各个模块的正确性。

4. 最少共享机制原则

避免多个主体共享同一个资源，因为敏感信息可能通过相同的机制在这些主体之间共享，导致被其他用户获取。每个主体应该有不同的机制或不同的机制实例，在保证多用户访问的灵活性的同时，应防止由单一的共享机制导致潜在的违背安全性的行为。

5. 完全中立原则

每次主体对资源进行请求时，系统都应该进行认证和检查，特别是和安全相关的内容，以避免错误地赋予主体过高的权限或者在第一次授予主体权限后，主体被攻击，攻击者滥用相关权限。为了提高性能，一些系统会临时存储主体的权限，这种做法易使系统具有较高的安全风险。

6. 心理可接受度原则

安全机制应该尽可能对用户透明，只引入少量的资源使用限制，对用户友好，才能方便用户的理解和使用，真正起到安全防护的作用。安全机制不应降低资源的可用性或使得资源难以获取，否则，用户很可能会选择关闭这些安全机制或功能。

7. 默认故障处理保护原则

任何复杂的系统都有发生故障的时候，这是很难避免的。当系统失效或发生故障时，必须以安全的方式来处理系统信息，系统故障处理应该是安全的。例如，即使丧失了可用性，也应该保障系统的机密性和完整性；故障发生时必须阻止未授权的用户获得访问权限；发生故障后，应该不向远程未授权的用户暴露敏感信息，如错误号和错误信息、服务器信息等。

8. 不信任原则

开发者应该假定系统环境是不安全的。对外部实体所有的输入都需要进行检查，即使输入来自可信的外部用户。另外，不应该认为每次对函数或系统的调用操作都必然会成功，如内存的分配，因此必须对每次函数或系统调用的返回值进行检查，并进行正确的处理。

9. 纵深防御原则

纵深防御原则是指：不仅依靠操作系统提供的安全防护机制，而且建立协议层次、信息流向等纵向结构层次的多种有效防御措施，形成纵深防御体系。由于攻击者要绕过每一个机制才能达到目的，纵深防御体系提高了攻击者的攻击成本和要求，降低了攻击者成功攻击的概率和危害。纵深防御的基本思想是：使用多重防御来管理风险，以便在一层防御无效的时候，另一层防御将会阻止入侵破坏。

10. 保障最薄弱环节原则

安全社区有一个常见的比喻：安全性是根链条，其强度取决于最薄弱的环节。攻击者一般从系统最薄弱的环节发起攻击，而不是针对已经加固的组件。相对于破解一个数学上已经证明了安全性的算法，攻击者更喜欢利用软件的安全漏洞。因此，软件开发者必须了解自己开发的软件的薄弱点，针对这些薄弱点实施更强的安全保护措施。

11. 公开设计原则

公开设计原则假定攻击者有能力获取系统足够的信息以发起攻击。如果加密算法存在弱密钥，或者系统设有万能口令，攻击者通过反汇编分析能够获取这些信息。攻击者还可能是内部被辞退的员工。因此，如果认为攻击者无法掌握某些特定信息来实施攻击，就可能给系统带来安全隐患。

12. 隐私保护原则

系统对其收集到的用户信息都必须实施妥善和安全的保护。攻击者获得了用户的隐私信息之后，可以进一步发起针对用户的各种攻击，如欺骗等。因此，不应该向其他用户泄露用户的隐私信息。

13. 攻击面最小化原则

攻击面(attack surface)，通常也称受攻击面，是指对一个软件系统可以采取的所有可能的攻击方法。因为攻击者对软件的攻击是通过其暴露在外部的接口、功能、服务和协议等资源实施的，所以，计算每种资源被成功攻击的可能性，并将这些可能性综合归纳，就可以衡量软件的攻击面大小。可以看出，一个软件的攻击面越大，其安全风险也就越大。

减小攻击面是安全设计中的一个重要步骤。软件设计人员需要仔细评估软件中所有功能模块和接口的特性，分析其可能存在的安全风险，并设定相应的限制措施。如果一个功能模块和接口不是必要的，则应该取消或禁止，或者默认不开启；如果没有特殊的理由，一个功能模块和接口默认应该按安全的方式进行设置。

2.4 安全设计方法

安全需求解决的是“做什么”的问题，而安全设计解决的是“怎么做”的问题。软件安全设计的目标是确保软件产品中不存在直接引起或间接促使系统趋于危险状态的构件。如果系统已经进入危险状态，检测出系统的危险状态和排除危险、采取对策的部件应能正

常工作;如果事故已经发生,控制并减轻损失的部件应能发挥作用。

2.4.1　危险性分析

在进行可靠性、安全性设计之前,要进行危险性分析,以确定软件安全关键单元。危险性分析是通过对系统存在的危险类别、出现条件、事故后果等条件进行分析,判断系统可能的潜在危险。危险性分析一般在软件编码之前完成,其分析结果将提交给关键设计评审(Critical Design Review,CDR)作为参考。

危险性分析的步骤如下:

(1) 根据需求危险性分析、概要设计危险性分析确定的危险事件,分析这些事件与低层次软件单元的关系,将对危险事件有影响的单元确定为软件安全关键单元,分析这些单元对危险事件施加影响的方式和途径。

(2) 在低层次上考察软件的各个单元、表、变量之间的相关程度,将直接和间接影响软件安全关键单元的其他单元确定为安全关键单元,分析它们对安全的影响。

(3) 分析软件安全关键单元的详细设计是否符合安全性设计的要求,分析的结果应送交软件设计人员和项目主管。确定在测试计划、说明和规程中需要包含的安全性要求。

(4) 确定在系统操作员手册、软件用户手册、系统诊断手册及其他手册中需要包含的安全性要求。

(5) 确保编程人员了解软件安全关键单元,向编程人员提供有关安全性的编程建议和要求。

2.4.2　基于模式的软件安全设计

近年来,模式方法在多个领域被广泛应用于解决各种通用问题。软件安全模式是指描述一个在特定环境中重复出现的特定安全问题,并为之提供一个良好的通用解决方案。基于模式的软件安全设计通过组合以往的软件设计模式来开发新的框架,以实现设计过程的软件复用,提升设计质量,加快设计效率。

结合 Open Group 的《安全设计模式》中总结的安全模式和软件设计方法,可以将常用的安全模式分为架构级模式、设计级模式和实施级模式。

1. 架构级模式

架构级模式可以分为不信任分解、特权分离和推迟到内核(Defer to Kernel)3 种类型。

不信任分解模式的目的是让独立的功能模块相互之间不信任,从而减少系统独立程序的攻击面,并且当有相互不信任的程序被攻破时,暴露给攻击者的功能和数据尽可能最少。

特权分离模式的目的是在不影响或限制程序功能的前提下,减少以特权权限运行的代码数量。

推迟到内核模式是为了清晰地分离需要提升特权的功能和不需要提升特权的功能,并使用现有的用户验证内核功能,利用内核已经建立的仲裁安全决策,而不是在用户级使

用仲裁安全决策的方法。

2 设计级模式

设计级模式可以分为安全工厂(Secure Factory)模式、安全策略工厂(Secure Strategy Factory)模式、安全构建器工厂(Secure Builder Factory)模式、责任安全链(Secure Chain of Responsibility)模式、安全状态机(Secure State Machine)模式和安全访问者(Secure Visitor)模式等。

在安全工厂模式下,访问者可以给予合法对象一个安全凭证;反过来,安全工厂模式能够针对已经给定的安全凭证选择和返回对应的对象。

安全策略工厂模式的目的是选择适当的策略对象来执行基于用户安全凭证的任务。它是安全工厂模式的扩展。

安全构建器工厂模式的目的是从创建对象的基本步骤里分离出涉及创建复杂对象的操作。

责任安全链模式的目的是在应用要求的功能函数中分离出用户信赖或者环境信赖的功能函数,并且简化这些功能。

安全状态机模式的目的是将安全性功能函数和用户级功能函数实现为两个单独的状态,使安全机制和用户功能清晰地分离。

安全系统可能需要在层次结构化数据上执行各种操作,数据层次结构中的每个节点可能有不同的访问限制,不同用户的数据访问权限依赖于角色/安全凭证。安全访问者模式允许给节点加锁,以免它被非法访问,除非访问者提供正确的安全凭证给节点解锁。

3 实施级模式

实施级模式主要有安全日志(Secure Logger)模式、清除敏感信息(Clear Sensitive Information)模式、安全目录(Secure Directory)模式、路径名规范化(Pathname Canonicalization)模式、输入验证(Input Validation)模式、资源获取初始化(Resource Acquisition Initialization,RAI)模式等。

安全日志模式能防止攻击者从系统日志中收集有用的系统信息,阻止攻击者通过编辑系统日志隐藏自己的行为。

清除敏感信息模式是指在被释放的资源可重用之前清除敏感信息,防止存储在一个可重用资源中的敏感信息被用户或对手未经授权访问。该模式确保存储在可重用资源中的敏感信息在重用之前被彻底清除。

安全目录模式确保攻击者无法操作程序执行期间使用的文件。

路径名规范化模式保证所有被程序读或写的文件通过一个有效的、不包含任何符号链接或快捷方式的路径(也就是规范化的路径名)访问。

输入验证模式是指验证输入数据的正确性。它要求开发人员正确识别和验证所有外部输入的不可信数据源。

资源获取初始化模式通过执行处理资源分配和回收对象的构造、析构等函数,保证系统资源在所有可能的程序执行路径下被合理地分配和收回。

2.4.3　安全关键单元的确定和设计

安全关键单元被定义为其错误可能导致系统潜在严重危险的软件单元。在确定安全关键单元时，首先需要在软件总体设计中确定安全关键部件，对这些部件进行细化和分解，再对产生的软件单元作进一步分析，最后从中确定安全关键单元。

通常考虑将下述软件单元确定为安全关键单元：

(1) 故障检测的优先级结构及安全性控制或校正单元，处理和响应故障的软件单元，中断处理程序、中断优先级管理及允许或禁止中断的例行单元。

(2) 产生对硬件进行自主控制的信号的单元。

(3) 产生直接影响硬件部件运动或启动安全关键行为的信号的单元。

(4) 其输出是显示安全关键硬件的系统状态的单元。

对于已经确定的安全关键单元，应在设计时遵循以下准则：

(1) 分离安全关键单元。尽量将安全关键单元与非安全关键单元分离，设立为不同的开发任务，并采取措施使后者的故障不会影响前者。

(2) 安全关键单元至少受控于两个独立的单元。

(3) 安全关键单元必须同一切其他单元隔离。安全关键单元必须放在一起，以便对其进行保护，并防止其他单元的干扰。

(4) 安全关键单元必须具有强数据类型。不得使用一位的逻辑 0 或 1 来表示“安全”或“危险”状态，安全关键状态的判定条件不得依赖于全 0 或全 1 的输入。

(5) 安全关键的计时单元必须由系统控制，不允许随意修改。

2.5　威胁建模

2.5.1　威胁建模概述

威胁建模是一种工程技术，它以结构化的方式识别、评估应用系统面临的威胁。威胁建模可使开发人员在设计阶段充分了解各种可能的安全威胁，对可能的风险进行管理，并指导开发人员选择适当的应对措施，根据设计和测试情况对安全架构和设计进行验证，从而降低软件的攻击面。通过执行威胁建模来确定何时何地需要资源以减少风险。在发现威胁的过程中，会发现许多可能的漏洞、威胁和攻击，但实际应用程序不太可能遇到所有这些漏洞，组织也不太可能需要解决所有这些问题，威胁建模可帮助开发人员识别组织应用中需要防范的威胁。

威胁建模也是一种风险管理模型，适用于所有的软件产品和系统的开发，帮助制订衡量安全目标的工程决策与其他设计目标。一个开发团队首先需要明确项目需要保护的目标，了解有哪些威胁和漏洞能够影响保护目标，找到缓解这些威胁和漏洞的具体措施，才有可能开发出安全的软件。威胁建模在软件生命周期的需求设计阶段就介入，并进行全面的威胁分析，通过消除常见漏洞来提高工程质量，使得安全防护成本更低，避免在软件

开发后期进行成本高昂的补救。威胁建模允许系统安全人员描述安全漏洞的破坏力，并按轻重缓急实施补救措施。

威胁建模不是一蹴而就的过程，需要重复进行，从应用程序设计的早期阶段开始，持续经过应用程序的整个生命周期。因为安全人员很难一次找出所有威胁，并且目前的应用程序很少为静态的，所以威胁建模过程需要随着应用程序的发展不断重复、持续改善并适应变化的应用程序。

威胁模型包括应用程序体系结构的定义和应用程序方案的一系列威胁，主要包括三大元素：资产、威胁和漏洞。资产是指需要保护的有价值的数据和设备，威胁是指攻击者对系统可能实施的攻击行为，漏洞是指系统内部可能被攻击者利用而对系统构成威胁的缺陷。以房子类比，房子中的珠宝是资源，窃贼是攻击者，门是房子的一部分，则开着的门表示一个缺陷，窃贼可以利用开着的门进入房子偷珠宝，即攻击者利用缺陷来获取资源。

在开始威胁建模过程之前，需要理解以下基本术语：

- 资源：指重要资源，如数据库中或文件系统中的数据以及系统资源。资源可能是信息或信息本身的可用性，例如客户数据；也可能是无形的，如公司的声誉。
- 威胁：是一种事件或潜在的事件，通常被描述为可能损害资产或目标的效果，它本质上可能是恶意的，也可能是非恶意的，可能会损害或危及资源的安全。
- 缺陷：是信息系统、信息技术或信息产品在某方面的弱点或特性，这些弱点可被攻击者利用以实施攻击，造成威胁的发生。网络、主机或应用程序中都可能存在缺陷。
- 攻击（或利用）：对某人或者某物采取的危害资源的行为。该行为可能是某人通过跟踪、威胁或者利用缺陷而实施的。
- 对策：应对威胁、减小危险的安全措施。对策包括改进应用设计、改进代码、改进操作实践等。

2.5.2 威胁建模过程

威胁建模的过程如图 2-3 所示，该过程可用于目前正在开发的应用程序以及现有的应用程序。

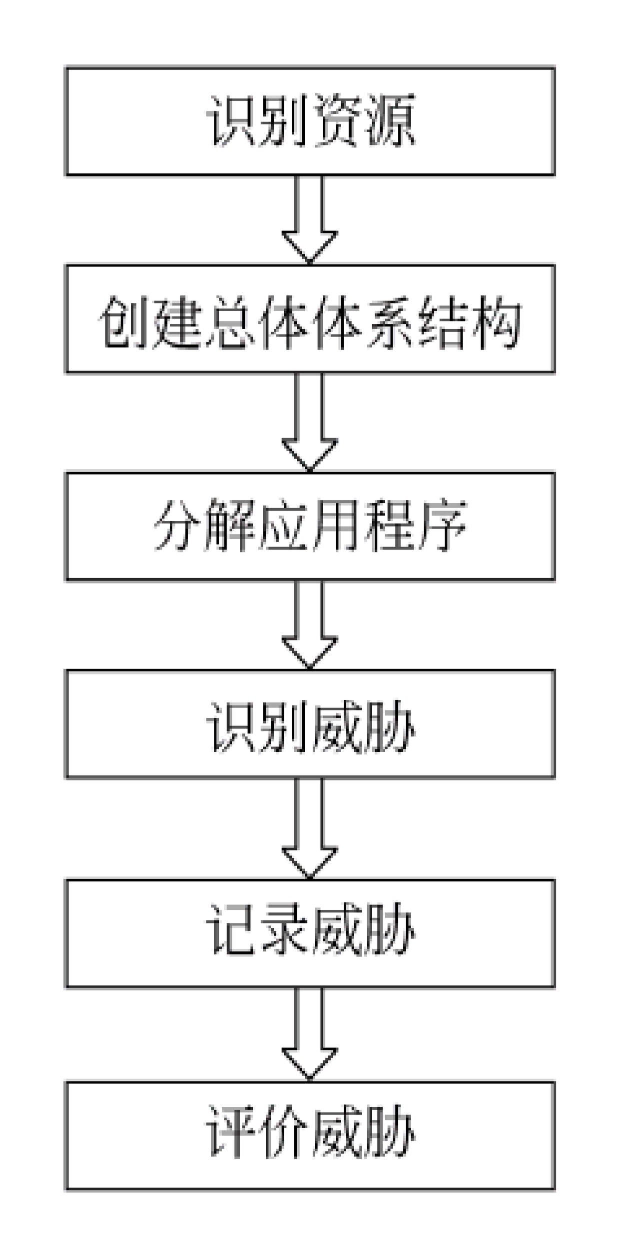


图 2-3　威胁建模过程

1. 识别资源

识别资源是指找出系统必须保护的有价值的资源，包括入口点和出口点、系统资产和资源、信任级别（访问类别），从如客户或订单数据库等机密数据到 Web 页或 Web 站点的可用性。

2. 创建总体体系结构

创建总体体系结构是指利用简单的图表来记录应用程序的体系结构，包括子系统、信任边界和数据流。这个阶段的目标是记录应用程序的功能、体系结构、物理部署配置及组成解决方案一部分的技术，寻找应用程序设计和实现中潜在的缺陷。

在这个阶段执行下列任务：

(1) 确定应用程序做什么以及它是如何利用和访问资源的。

(2) 创建高级的体系结构示意图。说明应用程序、子系统以及物理部署特点的组成和结构，如模拟与外部系统进行交互的示意图。

(3) 确定正确、恰当地用于实现解决方案的技术。最常用的技术包括 ASP. NET、Web 服务、企业服务、Microsoft . NET Remoting 以及 ADO. NET，还可以找出应用程序调用的任何非托管代码。

3. 分解应用程序

分解应用程序的体系结构包括基本的网络和主机基础结构的设计，根据传统的缺陷区域为应用程序创建安全配置文件。安全配置文件的目的是发现应用程序的设计、实现或部署配置中的缺陷。之后还要找出应用程序的信任边界、数据流、入口点和特权代码。对应用程序结构了解得越多，就越容易发现威胁。

在这个阶段执行下列任务：

(1) 标识信任边界。

找出包围应用程序的各实体资源的信任边界，这些资源由应用程序设计确定。对每个子系统，要考虑上游数据流或用户输入是否可信，如果不可信，要考虑对数据流和输入如何进行身份验证和授权。还要考虑调用代码是否可信，如果不可信，考虑如何对其进行身份验证和授权。必须能够保证适当的网关守卫程序可以将所有入口点都收敛或汇聚在一个特定的信任边界中，而且接收者的入口点要全面验证通过信任边界的所有数据。

(2) 标识数据流。

一种简单的标识方法就是以最高层作为起点，通过分析单个子系统间的数据流，以迭代的方式分解应用程序。跨信任边界的数据流特别重要，对于从信任边界外部传递数据的代码，应假定这些数据是恶意的，并对数据进行彻底验证。

另外，理解数据流图(Data Flow Diagram，DFD) 和序列示意图有助于安全人员规范地分解系统。DFD 是数据流、数据存储区和数据源与目标之间关系的图形化表示。序列示意图说明一组对象在有时间先后顺序的事件中是如何协作的。

(3) 标识入口点。

应用程序的入口点也是攻击的入口点。入口点可能包括侦听 HTTP 请求的前端 Web 应用程序，该入口点是有意公开给客户端的。其他入口点，如由跨应用程序层的子组件公开的内部入口点，它们的存在是为了支持与其他组件的内部通信。但是，系统安全人员应了解这些入口点位于何处，以及它们接收的输入类型是什么，以防攻击者设法绕过应用程序的封装、调用或引用而直接攻击内部入口点。对于每个入口点，安全人员还应能确定网关守卫程序的类型，对该程序进行授权并确定验证的级别和程度。

逻辑应用程序入口点包括由 Web 页提供的用户界面、由 Web 服务提供的服务界面、服务组件和. NET Remoting 组件以及提供异步入口点的消息队列。物理或平台入口点包括端口和插槽。

(4) 标识特权代码。

特权代码访问特定类型的安全资源,并执行其他特权操作。安全资源类型包括 DNS 服务器、目录服务、环境变量、事件日志、文件系统、消息队列、性能计数器、打印机、注册表、套接字和 Web 服务。特权操作包括非托管代码调用、反射、串行化、代码访问安全性权限以及对代码访问安全性策略的操纵。

特权代码必须通过代码访问安全策略被授予适当的代码访问安全权限,保证它所封装的资源和操作不会公开给不可信的和可能有恶意的代码。例如,.NET Framework 代码访问安全机制通过执行堆栈审核来检查授予调用代码的权限。但是,有时覆盖该行为并绕过堆栈审核是必要的,例如,当想用沙箱限制特权代码或分离特权代码时。但这样就公开了代码,有可能诱发恶意攻击。恶意代码可以通过可信的中间代码调用代码。覆盖代码访问安全权限默认的安全行为时要认证,并要采取适当的安全措施。

(5) 记录安全配置文件。

确定输入验证、身份验证、授权、配置管理以及其他应用程序最容易受到攻击的区域等使用的设计和实现方法,为应用程序创建一个安全配置文件。

4. 识别威胁

牢记攻击者的目标,利用对应用程序的体系结构和潜在缺陷的了解,找出可能影响应用程序的威胁。

还可以使用以下两种基本方法来识别威胁:

(1) 利用 STRIDE 模型来识别威胁。微软公司提出使用 STRIDE 模型来进行威胁建模,STRIDE 由 Spoofing(假冒)、Tampering(篡改)、Repudiation(抵赖)、Information Disclosure(信息泄露)、Denial of Service(拒绝服务)和 Elevation of Privilege(权限提升)的第一个字母组合而成,代表这 6 类主要的威胁,如表 2-1 所示。

表 2-1 STRIDE 模型针对的 6 类主要威胁

威胁	定 义	举 例
假冒	模仿其他人或实体	伪装成 microsoft.com 或 ntdll.dll
篡改	修改数据或代码	修改硬盘、DVD 或网络数据包中的 DLL
抵赖	声称没有执行某个操作	“我没有发送过那封电子邮件。”“我没有修改过那个文件。”“我确实没有访问过那个网站。”
信息泄露	把信息披露给那些无权知道的人	允许某人阅读 Windows 源代码;公布某个 Web 网站的用户清单
拒绝服务	拒绝为用户提供服务	使得 Windows 或 Web 网站崩溃,发送数据包并耗尽 CPU 时间,将数据包路由到某个黑洞中
权限提升	获得非授权访问权	允许远程因特网用户执行命令,让受限用户获得管理员权限

这 6 类主要威胁和系统安全属性是密切相关的,理解威胁也就是理解软件系统的安全目标。这些威胁影响的安全属性如表 2-2 所示。

表 2-2　6 类主要威胁影响的安全属性

威胁	受影响的安全属性	说　　明
假冒	可鉴别性	鉴别用户身份是否合法和正确
篡改	完整性	数据和系统资源只限适当的人员以适当的方式进行更改
抵赖	不可抵赖性	用户(合法或非法)不能否认自己的行为和与行为关联的内容
信息泄露	机密性	资源只限拥有权限的人员访问
拒绝服务	可用性	系统在需要时一切就绪,可以正常执行操作
特权提升	授权	明确允许或拒绝用户访问资源

使用 STRIDE 模型进行威胁建模,应按照确定建模对象、识别威胁、评估威胁以及消除威胁 4 个步骤反复循环进行,直到所有的威胁所带来的风险都在可接受范围之内。

(2) 使用分类的威胁列表。首先将威胁根据网络、主机和应用程序的种类进行分组,形成威胁列表。然后将威胁列表应用到应用程序体系结构中,对照检查早期发现的任何缺陷,排除无关的威胁。

在这个阶段执行下列任务:

(1) 识别网络威胁。

这是网络设计者和管理员的任务。分析网络拓扑、数据包的流动,以及路由器、防火墙和交换机配置,寻找潜在的缺陷,还要注意虚拟专用网(Virtual Private Network,VPN)端点。

在这一步要考虑的主要网络威胁如下:

- 利用依赖发送方 IP 地址的安全机制。
- 通过未加密网络通道传送会话标识符或 cookie,可能导致 IP 会话被劫持。
- 通过未加密通信通道传送明文的身份验证凭据或其他敏感数据。这可能会导致攻击者监视网络,获取合法用户的身份验证凭据,或者获取并篡改其他敏感数据。
- 来自不安全设备和服务器配置的威胁。应考虑以下问题:多余的端口被关闭了吗?多余的协议被禁用了吗?对路由表和 DNS 服务器进行保护了吗?对服务器上 TCP 网络堆栈进行强化了吗?

(2) 识别主机威胁。

配置主机安全(即 Microsoft Windows 2000 和.NET Framework 配置)时,将该配置分为单独的几类,这样就能够以结构化和逻辑的方式应用安全设置。理论上,该方法也适用于检查安全、确定缺陷以及识别威胁。适用于所有服务器角色的常用配置项目包括补丁和更新、服务、协议、账户、文件和目录、共享、端口、审查和日志记录。对于每一个配置项目,都要检查可能的易受攻击的设置,从这些设置中识别威胁。

在这一步要考虑的主要主机威胁如下:

- 未打补丁的服务器可能遭受病毒、特洛伊木马、蠕虫和 IIS 攻击。
- 使用非必需的端口、协议和服务。这将增大攻击面,使攻击者可以收集信息并攻击系统。

- 允许未经身份验证的匿名访问。
- 使用脆弱的密码和账户策略。这可以导致攻击者破解密码、盗用身份和发动拒绝服务攻击。

(3) 识别应用程序威胁。

使用 STRIDE 模型和预定义的威胁列表来仔细检查应用程序的安全配置文件的各个方面,集中考虑应用程序威胁、技术特有的威胁和代码威胁。

在这一步要考虑的主要应用程序威胁如下:

- 使用的输入验证不当将导致跨站脚本攻击、SQL 注入攻击和缓冲区溢出攻击。
- 在未加密的网络链路上传送身份验证凭据或者身份验证 cookie,会导致攻击者捕获身份验证凭据或者劫持会话。
- 使用脆弱的密码和账户策略可能导致非授权访问。
- 保护应用程序的配置管理(包括界面管理)不当。
- 以明文的形式存储配置信息,例如连接字符串和服务账户凭据。
- 越权使用进程和服务账户。
- 使用不安全的数据访问编码技术,这可能会增大 SQL 注入攻击的威胁。
- 使用脆弱的或者自定义的加密技术,并且未能充分保护密钥。
- 完全依赖从 Web 浏览器传递过来的参数,例如窗体字段、查询字符串、cookie 数据以及 HTTP 协议头。攻击者可以冒用其他用户身份,或者在提交的内容中注入恶意代码。
- 使用不安全的异常处理机制,这可能导致拒绝服务攻击或者泄露对于攻击者来说非常有用的系统级详细资料。
- 审核与日志记录不充分,这可能导致抵赖。

5. 记录威胁

利用通用威胁模板记录每一种威胁。该模板定义了一套要捕获的各种威胁的核心属性。威胁描述和威胁目标是威胁的基本属性,其他属性还有攻击方法。

6. 评价威胁

在威胁建模过程的最后阶段对确定的威胁列表进行评价以区分优先顺序。首先处理最重要的威胁,因为这些威胁带来的危害最大。在评价过程中要权衡威胁的可能性以及攻击发生时可能造成的危害。评价的结果可能是:通过对比威胁带来的风险和为消除威胁所花费的成本,对于某些威胁采取行动是不值得的。

$$风险=威胁发生的概率\times潜在的损失$$

该公式表明,特定威胁造成的风险等于威胁发生的概率乘以潜在的损失,这表明了如果攻击发生将会对系统造成的后果。

简单的评价系统的问题在于小组成员对评价结果通常意见不一。为解决这个问题,可以使用微软公司的 DREAD 模型来计算威胁的严重性。通过确定下列项目的等级范围,就可以得到威胁的风险评价结果。

- 破坏潜力(Damage potential):如果漏洞被利用,损失有多大?

- 再现性(Reproducibility)：重复被利用的难度有多大？
- 可利用性(Exploitability)：漏洞被利用的难度有多大？
- 受影响的用户(Affected users)：多少用户可能受到影响？
- 可发现性(Discoverability)：漏洞容易被发现吗？

DREAD 由上述 5 个项目的英文首字母组成。

对大多数项目来说，等级范围为 1～3，依次从严重到危害小。

2.5.3　威胁建模的输出与缓解

威胁建模过程的输出是向项目成员提交一份工作项目报告，内容包括应用程序体系结构安全方面的记录和评价过的威胁列表。该工作项目报告可使项目成员更加清楚地了解需要进行处理的威胁，以及如何对其进行处理。设计者可以利用它来进行技术与功能方面的安全设计决策；编写代码的开发人员可以利用它来降低风险；测试人员可以编写测试用例来测试应用程序是否容易受到这些威胁的攻击。在报告中，根据网络、主机和应用程序种类来组织威胁。这可以使不同角色的不同小组成员更方便地使用该报告。在每一类中，按优先顺序排列威胁，最前面的是评价具有最高风险的威胁，其后是危险较小的威胁。

根据威胁的评估结果，确定是否要消除或缓解该威胁的技术措施。在设计阶段，可以通过重新设计来直接消除威胁，或采用技术手段来消除威胁。在本阶段，应在确定消除或缓解威胁的措施后继续评估是否可以接受残余的安全风险。

消除和缓解威胁的手段在计算机软件中是多种多样的，常用的技术如下：

(1) 在处理可能来自不可信来源的数据时，安全软件必须验证其输入。

(2) 开发人员将应用程序放在沙盒中运行，最小化应用程序的损失。

(3) 划分应用程序并确保应用程序的每个部分只能访问它所需要的信息，使信息泄露风险最小化。

(4) 进行模糊测试，向应用程序或守护进程发送异常数据，查看其运行是否中断，修复在这个过程中发生的任何缺陷。

(5) 采用内置安全功能的操作系统。

(6) 根据风险大小，选择适当的方法或技术来缓解每一种威胁。威胁缓解措施必须结合软件系统的实际情况提出。可供参考的一些技术措施包括：

- 假冒。使用认证技术来解决，如 cookie 认证、Kerberos 认证或数字签名等技术。
- 篡改。使用数据完整性技术来解决，如采用哈希函数、消息认证码、数字签名等密码算法或防篡改协议来传输数据。
- 抵赖。使用非抵赖性服务技术来解决，如强认证、安全审计、数字签名、时间戳等技术。
- 信息泄露。使用数据保护技术来解决，如采用对称加密、非对称加密等密码算法，或采用访问控制手段限制访问，或采用隐私保护协议保护隐私数据的存取等技术。
- 拒绝服务。使用提高系统可用性技术来解决，如对服务对象采用过滤、认证、授

权、配额等技术对合法用户访问量进行控制,采用流量控制技术来保护系统的稳定运行,采用负载均衡技术来提高系统整体服务能力等。

- 特权提升。使用严格授权和限制访问技术来解决,如采用访问控制列表、授权管理等技术对用户进行严格授权,同时要求软件系统以最低权限运行,各模块设计以最小权限执行,等等。

第 3 章 C 和 C++安全编码

随着信息通信技术和互联网技术的飞速发展,网络不仅改变了人们的生活方式,还推动了传统工业、新兴服务业和信息产业的快速发展,带动了国民经济发展和社会进步。互联网逐渐改变了人们的生活方式,而 C 和 C++ 凭借其强大的底层操作能力,在网络的飞速发展过程中扮演了一个极其重要的角色。本章介绍 C 和 C++ 开发安全现状、C 和 C++ 常见安全漏洞以及 C 和 C++ 安全编码规范。

3.1 C 和 C++ 开发安全现状

TIOBE 编程语言排行榜如图 3-1 所示。C 和 C++ 比其他的高级语言拥有更强大的底层操作能力,因而具有功能强大、效率高的优点,各种操作系统的实现都大量使用 C 和 C++ 进行开发,对效率有较高要求的服务程序以及对响应速度有较高要求的客户端程序也大多使用 C 和 C++ 开发。C 和 C++ 的应用领域非常广泛,无论是操作系统、浏览器、嵌入式开发还是游戏引擎、各类编辑器开发,C 和 C++ 都占有非常大的市场份额。

Oct 2019	Oct 2018	Change	Programming Language	Ratings	Change
1	1		Java	16.884%	-0.92%
2	2		C	16.180%	+0.80%
3	4	⌃	Python	9.089%	+1.93%
4	3	⌄	C++	6.229%	-1.36%
5	6	⌃	C#	3.860%	+0.37%
6	5	⌄	Visual Basic .NET	3.745%	-2.14%
7	8	⌃	JavaScript	2.076%	-0.20%
8	9	⌃	SQL	1.935%	-0.10%
9	7	⌄	PHP	1.909%	-0.89%
10	15	⌃⌃	Objective-C	1.501%	+0.30%

图 3-1 TIOBE 编程语言排行榜

C 和 C++ 语言拥有强大的内存操控能力,具有指针、动态内存分配等特点,这些特点带给 C 和 C++ 语言强大的生命力。但是,因为 C/C++ 语言缺乏内存检查机制,因此比较容易出现非法指针解引用、内存泄漏等一系列安全漏洞,所以 C/C++ 语言不适合编写对安全性要求高的程序。

在 C 和 C++ 语言中，大多数缓冲区溢出问题可以直接追溯到标准 C 和 C++ 语言库，但是在用 C 和 C++ 语言编写的程序中仍然会调用这些存在漏洞的危险函数，因为开发人员往往不知道如何避免使用这些危险函数。即使有些开发人员获得安全提示，但是由于函数使用规则灵活多样，即使是优秀的开发人员也不能完美地避开这些安全陷阱。他们会在危险函数的自变量上使用自己编写的检查代码，或者错误地认为在某些特殊情况下使用有潜在危险的函数是安全的。

使用 C 和 C++ 语言编码的程序员学习和关注的大都是程序的逻辑性和可用性。在没有边界检查机制的编程语言里，逻辑可能会走错路，因为计算机可以访问和执行任意内存中的内容，而且访问的内容大多是和程序中的代码和变量没有关系的，所有没有边界检查的编程语言会将计算机的多个维度暴露给程序，因此极易引入漏洞，造成不同程度的危害。

心脏滴血漏洞就是 C 和 C++ 语言缺乏边界检查机制造成的缺陷，因为这个漏洞不是普通的被触发的动作，因此它无法被 Valgrind 这样的工具检查出来，而是需要通过恶意的行为或足够智能的测试协议才能发现，但这通常是十分困难的。

此外，C 和 C++ 语言不能够自动地进行边界检查，这样会导致一系列安全隐患。但同时，不进行边界检查使得 C 和 C++ 语言的效率得到大幅度的提升，因此边界检查的代价是程序的效率。一般来讲，C 和 C++ 语言在大多数情况下注重效率。然而，获得较高效率的代价是 C 和 C++ 语言程序员必须十分警觉并且有极强的安全意识，这样才能防止他们的程序出现问题，而且即使这样，使代码不出问题也不容易。很多开发人员对边界检查的危害了解得并不多，因此也更容易忽视代码安全开发。

3.2 C 和 C++ 常见安全漏洞

如果软件开发人员的安全编程能力和水平不够，在软件开发的过程中就会引入漏洞。以下是 C 和 C++ 常见的安全漏洞。

3.2.1 缓冲区溢出漏洞

缓冲区是指程序中定义的存储数据的一组地址连续的内存单元。缓冲区溢出是指从缓冲区读数据或者向缓冲区写数据时超出了缓冲区定义的容量，从而覆盖了缓冲区以外的其他内存数据的现象。缓冲区溢出是一个令人头疼的问题，因为在软件的开发和测试阶段并非总能发现该问题。

然而，并非所有的缓冲区溢出都会造成软件漏洞。如果攻击者能够通过操纵用户输入来利用安全缺陷，那么缓冲区溢出就会导致漏洞了。例如，有一些广为人知的技术可以用于覆写栈帧来执行任意的代码。缓冲区溢出也可以在堆或静态内存区域被利用，方法是覆写邻接内存单元的数据。

ASA Heartbleed 是最常见的心脏滴血漏洞，其 CVE 编号为 CVE-2014-0160，简化后的漏洞相关代码如下：

```
var subItem;
uint32 lengthToUin32Max = length.IsSmallIndex() ? length.GetSmallIndex() :
MaxArrayLength;
for (uint32 idxSubItem = 0u; idxSubItem < lengthToUin32Max; ++idxSubItem)
{
        if (JavascriptOperators::HasItem(itemObject, idxSubItem))
       {
              subItem = JavascriptOperators::GetItem(itemObject, idxSubItem, scriptContext);
                                                        将GetItem的返回值赋给subItem
              if (pDestArray){
                     pDestArray->DirectSetItemAt(idxDest, subItem);
              }
              else{
                     SetArrayLikeObjects(pDestObj, idxDest, subItem);
              }
       }
       ++idxDest;
}
```

漏洞修复方式如下：

```
hbtype = *p++;
n2s(p, payload);
if (1 + 2 + payload + 16 > s->s3->rrec.length)    检查payload的长度
   return 0; /* silently discard per RFC 6520 sec. 4 */
pl = p;
```

3.2.2　释放后使用漏洞

释放后使用(Use After Free,UAF)漏洞是指程序中错误地保留了对已释放的内存对象的引用,继续访问已释放的内存。这种漏洞在浏览器程序中出现得非常多,利用 UAF 漏洞进行挂马的网络攻击也层出不穷。这种漏洞原理和利用流程如图 3-2 所示。

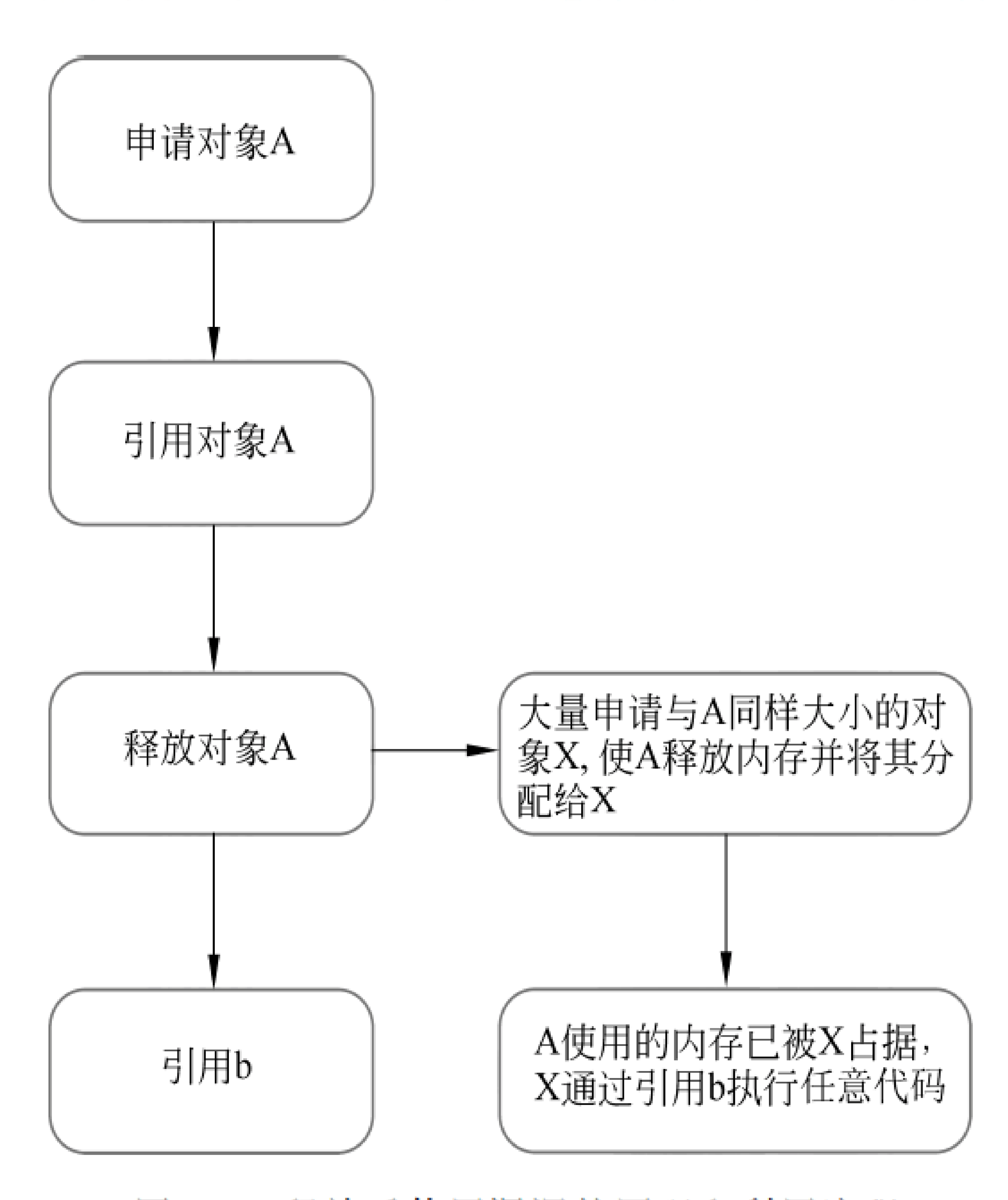


图 3-2　释放后使用漏洞的原理和利用流程

释放后使用漏洞在浏览器中的示例如下：

```
<!doctype html>
<!--Tested on win7 x86 IE9 version(9.0.8112.16555) with page heap enabled -->
<html>
<body onload="test();">
<script>
function randEventListener()
{
    document.write("475046")
}
function test()
{
    elm = document.createElement('title')
    document.body.appendChild(elm)
    document.attachEvent("onpropertychange", randEventListener)
    document.body.outerHTML = document.body.outerHTML
    o = document.all[0]
    o.textContent = ''
}
</script>
</body>
</html>
```

3.2.3 整型溢出漏洞

在计算机中，整数分为无符号整数以及有符号整数两种。其中有符号整数会在最高位(符号位)用 0 表示正数，用 1 表示负数，而无符号整数则没有这种限制。整数溢出就是将整型数据放入比它小的存储空间中，从而导致溢出。

一般来说，主要有以下 3 类整型溢出漏洞：

(1) 无符号整数的下溢和上溢。

编程语言定义的数据类型都有其数据取值范围。当超过这个取值范围时，其取值会从另一端的取值端点返回。例如，一个短整型数变量的取值范围是 0～65 535。当初始值为 65 535 时进行＋1 操作，该数值将返回至初始点，即 0，这种情况称为上溢；而当初始值为 0 时进行－1 操作，该数值将变为 65 535，这种情况称为下溢。

(2) 符号问题。关于这个问题，有以下 3 点需要注意：

- 有符号整数之间的比较。
- 有符号整数的运算。
- 无符号整数和有符号整数的比较。

(3) 截断问题。这个问题主要发生在将位数较多的整数(如 32 位整数)复制到位数较少的整数(如 16 位整数)时。

3.2.4 空指针解引用漏洞

指针变量可以指向堆地址、静态变量和空地址单元。空指针解引用是指当引用指向空地址单元的指针变量时，就会产生不可预见的错误，导致软件系统崩溃。空指针引用漏洞可能导致系统崩溃、拒绝服务等诸多不良后果。

空指针引用漏洞主要存在于操作系统、服务器应用程序等软件系统中。这些漏洞一

旦被恶意攻击者利用，就可能导致系统崩溃，服务器程序可能会拒绝服务，或者使机密信息泄露，这些都将严重地影响软件的运行以及系统的安全。

例如，Windows 针对传递给 Windows 内核系统调用的注册键值没有进行充分的校验。攻击者通过运行特殊构建的应用程序，使内核触发空指针引用漏洞而造成系统崩溃。Linux Kernel 是 Linux 所使用的内核，其 kernel/posix-timers. c 文件中的 clock_nanosleep()函数存在安全漏洞，如果使用等于 CLOCK_MONOTONIC_RAW 的时钟 ID 调用该函数，就会触发空指针引用漏洞，导致拒绝服务的情况。关闭套接字后重置没有阻止对已经废弃的套接字的操作，可造成空指针引用漏洞。

3.2.5　格式化字符串漏洞

格式化字符串漏洞指软件使用了格式化字符串作为参数，且该格式化字符串来自外部输入。会触发该漏洞的函数很少，主要有 printf()、sprintf()、fprintf()等函数。

在上述函数中使用的基本的格式化字符参数如下：

%c：输出字符，结合%n 可用于向指定地址写数据。

%d：输出十进制整数，结合%n 可用于向指定地址写数据。

%x：输出十六进制数据，如%i $ x 表示输出地址偏移量为 i 的 4 字节长的十六进制数据，%i $ lx 表示输出地址偏移量为 i 的 8 字节长的十六进制数据，这在 32 位和 64 位环境下是一样的。

%p：输出十六进制数据，与%x 基本一样，只是附加了前缀 0x，在 32 位环境下输出 4 字节，在 64 位环境下输出 8 字节，可通过输出字节的长度来判断目标环境是 32 位环境还是 64 位环境。

%s：输出字符串，如%i $ s 表示输出地址偏移量为 i 的地址所保存的字符串，这在 32 位和 64 位环境下是一样的，可用于读取 GOT 表等信息。

%n：将%n 之前 printf()已经打印的字符个数赋值给偏移处指针所指向的地址，如%100x10 $ n 表示将 0x64 写入偏移 10 的指针所指向的地址(4 字节)，而% $ hn 表示写入的地址空间为 2 字节，% $ hhn 表示写入的地址空间为 1 字节，% $ lln 表示写入的地址空间为 8 字节，这在 32 位和 64 位环境下是一样的。有时，直接写 4 字节会导致程序崩溃或等候时间过长，可以通过% $ hn 或% $ hhn 来适当调整。

%n 是通过格式化字符串漏洞改变程序流程的主要方式，而其他格式化字符串参数可用于读取信息或配合%n 写数据。

格式化字符串漏洞在利用时分为两种：

(1) 有二进制程序且格式化字符串在栈中。由于格式化字符串就保存在栈中，可以比较方便地利用%n 来写入数据以修改控制流。

(2) 有二进制程序且格式化字符串不在栈中。由于格式化字符串是保存在堆中的，不能像保存在栈中那样直接修改函数返回地址。此时，可以通过% $ n 实现堆栈互换，达到修改返回地址的目的。

3.2.6 内存泄漏

内存泄漏是指程序中已动态分配的堆内存由于某种原因未被释放或无法释放，造成系统内存的浪费，导致程序运行速度减慢甚至系统崩溃等严重后果。

内存泄漏按照产生的方式可以分为 4 类：

（1）常发性内存泄漏。发生内存泄漏的代码会被多次执行，每次被执行时都会导致一个内存区域泄漏。

（2）偶发性内存泄漏。发生内存泄漏的代码只有在某些特定环境或操作过程下才会被执行。常发性内存泄漏和偶发性内存泄漏是相对的。对于特定的环境，偶发性内存泄漏也许就变成了常发性内存泄漏。所以测试环境和测试方法对检测内存泄漏至关重要。

（3）一次性内存泄漏。发生内存泄漏的代码只会被执行一次，或者由于算法上的缺陷，导致总会有一个且仅有一个内存区域发生泄漏。

（4）隐式内存泄漏。程序在运行过程中不停地分配内存，直到运行结束的时候才释放内存。严格地说，这里并没有发生内存泄漏，因为最终程序释放了所有申请的内存。但是一个服务器程序往往需要运行几天、几周甚至几个月，不及时释放内存也可能导致最终耗尽系统的所有内存，所以称这类内存泄漏为隐式内存泄漏。

3.2.7 二次释放漏洞

二次释放（double free）是指程序分配一个内存区域之后，经过使用将这个内存区域释放，但并没有将指向这个内存区域的所有指针清零或回收，并在其他地方再次将指向同一个内存区域的指针交给内存分配器进行释放操作。该漏洞可能导致程序崩溃、信息泄露或任意代码执行等后果。此漏洞的原理和利用流程如图 3-3 所示。

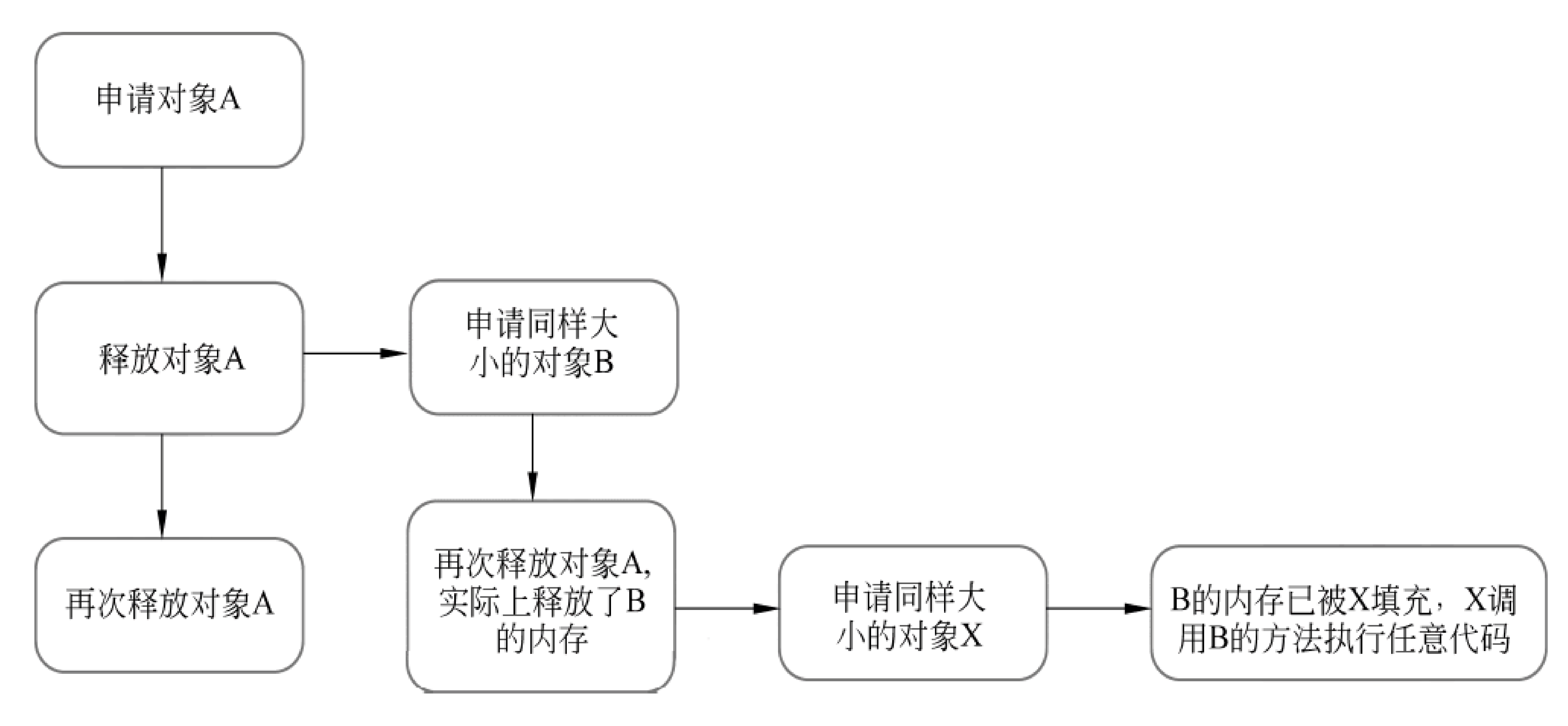


图 3-3　二次释放漏洞的原理和利用流程

以下是 Windows afd.sys 触发二次释放漏洞的过程：

（1）IOCTL 0x1207f（afd!AfdTransmitFile）会创建一个 TpInfo 结构，该结构中的 0x40 处是 TpInfoElement 数组，TpInfoElement 数组的 0x10 处指向 MDL（Memory

Descriptor List,内存描述符列表)结构。

(2) AfdTransmitFile 在返回前调用 afd!AfdReturnTpInfo 释放 MDL。

(3) 在 IOCTL 0x120c3(afd!AfdTransmitPackets)调用中存在一条路径,会再次释放之前已经释放的 MDL,造成系统崩溃。

3.2.8　类型混淆漏洞

类型混淆漏洞指程序将指针、对象等资源初始化为一种类型,随后以另一种(并不兼容的)类型对这些资源进行访问。

例如,微软公司曾修补了一个 CVE 编号为 CVE-2015-1641 的 Word 类型混淆漏洞。攻击者可以构造嵌入了 docx 的 rtf 文档进行攻击。Word 在解析 docx 文档,处理 displacedByCustomXML 属性时未对 customXML 对象进行验证,可以传入其他标签对象进行处理,造成类型混淆,导致任意内存写入。最终,经过精心构造的标签以及相应的属性值可以远程执行任意代码。

3.2.9　未初始化漏洞

未初始化漏洞是指对程序中变量或对象未赋予初始值就直接使用。未初始化的内存中存储的值是随机的,因此该地址处的变量值就是随机的,会对程序的运行造成不可预计的影响。在严重的情况下,未初始化漏洞可能导致信息泄露和任意代码执行。此漏洞的利用方法如图 3-4 所示。

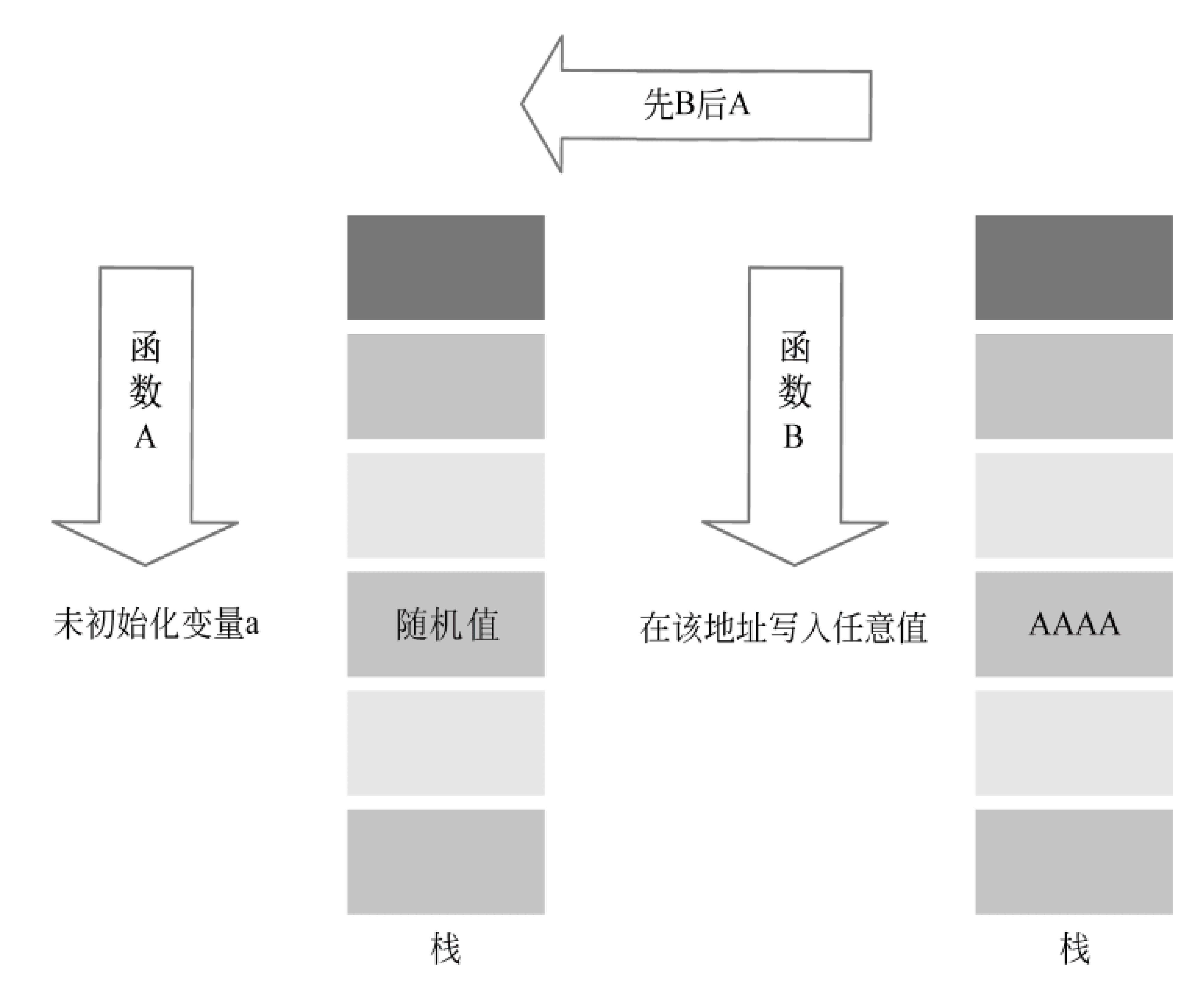


图 3-4　未初始化漏洞的利用方法

以 Chakra 引擎 concat 方法未初始化漏洞为例,与漏洞相关的代码如下:

```
var subItem;
uint32 lengthToUin32Max = length.IsSmallIndex() ? length.GetSmallIndex() :
MaxArrayLength;
for (uint32 idxSubItem = 0u; idxSubItem < lengthToUin32Max; ++idxSubItem)
{
    if (JavascriptOperators::HasItem(itemObject, idxSubItem))
    {
        JavascriptOperators::GetItem(itemObject, idxSubItem, &subItem, scriptContext);
        if (pDestArray)
        {
            pDestArray->DirectSetItemAt(idxDest, subItem);
        }
    }
}
```

上述代码未对 subItem 进行初始化，因此造成未初始化漏洞。修复代码如下：

```
var subItem;
uint32 lengthToUin32Max = length.IsSmallIndex() ? length.GetSmallIndex() :
MaxArrayLength;
for (uint32 idxSubItem = 0u; idxSubItem < lengthToUin32Max; ++idxSubItem)
{
    if (JavascriptOperators::HasItem(itemObject, idxSubItem))
    {
        subItem = JavascriptOperators::GetItem(itemObject, idxSubItem, scriptContext);
                                            将GetItem的返回值赋给subItem
        if (pDestArray){
            pDestArray->DirectSetItemAt(idxDest, subItem);
        }
        else{
            SetArrayLikeObjects(pDestObj, idxDest, subItem);
        }
    }
    ++idxDest;
}
```

3.3 C 和 C++ 安全编码规范

主要的 C 和 C++ 安全编码规范如下：

(1) 在代码中不要直接写入明文口令、密码、身份证等用户敏感信息。

(2) 不要通过日志打印明文密码等敏感信息。为了方便地跟踪程序运行过程中的信息，通过日志打印信息是程序员编写代码时常用的一种调试方式。应避免在日志中打印明文密码等敏感信息。

(3) 对不安全 C 和 C++ 库函数进行重写后再调用。以 C++ 为例，C++ 中的字符串函数均为不安全函数，当相关参数可以被外界控制时，就会导致溢出漏洞。重写这些函数的关键是在函数内部检查输入参数的合法性，例如，源字符串长度不超过目的字符串长度，最后一个\0 要进行特殊处理，等等。

(4) 定义了指针成员变量、静态成员变量的类必须重写拷贝构造函数。在 C++ 中，有 3 种情况需要调用拷贝构造函数：

① 对象以值传递的方式传递给函数参数。

② 对象以值传递的方式从函数返回。

③ 对象需要使用另一个对象进行初始化。

(5) 定义为基类的析构函数必须定义为虚函数。例如：

```
Class A{};
Class B : public A{};
A *a = new B();
```

如果不将类 A 的析构函数定义为虚函数，那么在通过 delete a 释放存储空间时，派生类的对象的存储空间不会被释放(派生类的析构函数不会被调用)。相反，如果不需要基类对派生类及对象进行操作，则不能将析构函数定义为虚函数，因为这样会增加内存开销。

(6) 注意线程安全。在多线程的环境下需要考虑并发、线程同步、互斥锁等相关概念，任何操作都会有其他线程竞争当前线程的资源。

(7) 对不安全加解密算法进行检查。不可逆加密算法 md5、sha1、RSA(小于 1024 位)和可逆加密算法 AES(小于 128 位)、DES(K1\K2\K3 至少有两个相同)均为不安全算法，要避免使用。同时，在不需要还原业务数据的情况下要使用不可逆加密算法。使用哈希算法时最好要加一个盐(salt)值。

(8) 注意整数操作安全。有符号整数操作要避免溢出，无符号整数操作要避免反转。以 32 位整数为例，有符号整数的取值范围是 $-2^{31} \sim 2^{31}-1$，无符号整数的取值范围是 $0 \sim 2^{32}-1$。对有符号整数和无符号整数进行操作时，要提前判断操作结果是否会超出合法的取值范围。例如，对有符号整数 num1 和 num2 进行相加操作时，需要判断：

```
if(num1 > MAX- num2)
```

如果条件成立，则计算结果会溢出。

第 4 章 Java安全编码

互联网的飞速发展使 Java 成为当今最流行的网络编程语言。无论是企业级应用还是面向大众的服务应用,基于 Java 的应用开发都变得越来越重要。然而,随着 Java 应用的普及和深入,各种安全漏洞不断出现。这也使得人们越来越重视编码安全问题。本章首先介绍 Java 开发的安全现状,其次对常见的 Java 安全漏洞进行分析,并提出防御措施,最后介绍 Java 安全编码规范,以指导开发人员进行安全编码。

4.1 Java 开发安全现状

随着互联网的发展,Java 语言的诸多优点引起了软件开发人员极大的关注。Java 以其面向对象、跨平台、安全性、多线程等特点而成为许多应用系统的理想开发语言。它可以用来进行面向对象的应用开发、可视化和可操作化的软件开发、动态画面的设计和调试、数据库的操作和连接设计等。作为现阶段最流行的网络编程语言,Java 在金融、电信、制造、在线电子商务软件等领域都得到了普遍应用。

Java 是由 Sun Microsystems 公司推出的程序设计语言和平台的总称。Java 语言是一种能够编写跨平台应用程序的计算机语言,具有结构清晰、语法简单等特点。Java 平台由 Java 虚拟机(Java Virtual Machine)和 Java 应用编程接口(Application Programming Interface,API)两部分构成。Java 虚拟机是运行所有 Java 程序的抽象计算机,是实现 Java 语言跨平台特性的关键。大多数语言需要先编译成目标代码,再在相应的平台上运行;而在 Java 中,由于已经嵌入了几乎所有的操作系统,因此 Java 程序只需编译一次,就可以不加任何修改地在各种系统中运行。Java API 是一些预先定义的函数,为 Java 程序提供了一个独立于操作系统的标准接口,使其无须访问源码或理解内部工作机制的细节,即可访问一组例程。

Java 在程序开发中应用十分广泛。Servlet/JSP 用来实现网页和 Java 语言的沟通;Hibernate、Spring、Struts 用来对程序进行架构设计,使程序架构清晰,易于分析和维护。

Java 可分为 3 种版本:

(1) 企业版本的 J2EE。为综合运用各大企业的外部环境以及市场服务中心而研发的一种计算机网络技术平台,主要用于应用程序设计,由 EJB、Servlet 等软件构成。

(2) 标准版本的 J2SE。主要面向普通用户市场,其应用领域包括图形界面编程、工具界面编程、Java 数据库编程等几个重要部分。

(3) 微型版本的 J2ME。此版本主要是为了降低 Java 的复杂度,可应用于手机、平板

电脑等各种无线设备。

过去十几年的网络爆炸性增长和现在人们对计算机网络互联的依赖，将网络安全需求提升到了一个新的层次。Java 作为一种适用于网络的语言，自然离不开网络，网络在让人们的计算机互联互通的同时，也隐藏着很多隐患。近些年，由 Java 引发的网络安全攻击占所有安全问题的大部分。

在目前的 Java 开发中，程序员往往会从第三方库导入成千上万行代码，这些导入的代码通常用于执行通用任务，如数据库访问、XML 处理、日志记录等。然而由于导入的代码大多是开源的，企业无法保证其安全性。如果引用的代码中存在安全漏洞，程序开发人员也没有对其进行安全检查，攻击者就可以利用导入的代码中的缺陷，通过 SQL 注入等方式入侵到系统内部，进行服务器攻击或数据访问。不安全的开源代码导入造成了企业对这些安全漏洞一无所知，而攻击者却对它们的现状了如指掌，因此保证 Java 应用的安全是一个艰巨的任务。

典型的由第三方库造成的安全漏洞是 2014 年初的“心脏滴血”事件。它是在 Open SSL 加密库中发现的安全漏洞，影响了互联网上 2/3 的 Web 服务器。虽然这个漏洞和 Java 无关，但依旧值得人们关注。

此外，安全信息提供商 Secunia 公司最近在安全公告中发布了一个新的高危等级的安全漏洞——Sun 公司在各种浏览器和操作系统中运行 Java 程序的一个插件，这个 Java 插件可使小型网络程序在用户计算机上运行。在该插件中发现的安全漏洞允许恶意网站绕过用户的安全措施，通过浏览器在用户计算机上自动运行恶意程序，造成大量计算机受害。由于该插件在各计算机上十分常见，因此由它产生的漏洞可被用来攻击 Windows 和 Linux 等各种常用操作系统，并且这一过程无须用户干预，其严重性不言而喻。

Java 漏洞之所以爆发得如此猛烈，与人们不常更新 Java 版本有关。虽然甲骨文公司频繁发布 Java 更新以便控制漏洞，且 Java 本身也包括自动升级功能，不过这种保护措施现在看来似乎效果不大。由于 Java 应用数量巨大，因此发布更新的时间往往不够及时，并且补丁常常会破坏应用软件的功能。Java 应用的安全防护通常基于网络或测试，但随着技术的发展，这两种方法都开始变得不够可靠。

基于网络的防御措施包括 Web 应用防火墙和运行时入侵防御系统，可以抵御外部威胁。为了避免阻塞合法的流量，这些系统需要进行微调以降低其有效性。程序员为此往往需要投入大量精力来调整，使之只允许合法流量通行，避免授权的流量都被阻塞，使真实用户也被拒之门外。

应用测试工具可以分析软件是否存在漏洞，但其分析结果的信息量往往十分庞大，让人难以区分哪些是关键问题，哪些是次要问题和误报。此外，如果漏洞侥幸入侵成功，这些工具在系统运行时便无法起到保护作用。

Java 的安全开发是一个艰巨的任务。以现在的安全技术来说，针对 Java 安全问题的建议仍是及时升级 Java，使用 IDS /IPS（Intrusion Detection System/Intrusion Prevention System，入侵检测系统/入侵防御系统）和及时更新签名，不要浏览不安全的网站，等等。真正针对 Java 的安全防御措施仍然缺口很大。

4.2 Java 常见安全漏洞

4.2.1 SQL 注入漏洞

注入(injection)类漏洞是应用系统中最常见的安全漏洞。SQL 是结构化查询语言(Structural Query Language)的简称,大多数应用都使用 SQL 数据库来存放应用程序的数据。由于 SQL 语法允许数据库命令和用户数据混杂在一起,攻击者可将恶意的 SQL 命令插入表单的输入域或页面请求的查询字符串中提交给服务器。如果应用程序没有对用户的输入进行检查和过滤,并在接收输入内容后将攻击者的输入作为原始 SQL 查询语句的一部分,则恶意输入将会改变程序原始的 SQL 查询逻辑,从而执行攻击者构造的任意命令。例如,很多影视网站泄露的 VIP 会员密码大多都是通过表单提交查询字符获得的。攻击者通过 SQL 注入攻击可以获取网站数据库的访问权限,从而获取网站数据库中的所有数据,篡改数据库中的数据甚至毁坏数据。

1. SQL 注入攻击方法

SQL 注入攻击方法可分为回显注入和盲注。基于错误回显的 SQL 注入是指利用 SQL 语句的矛盾性使得数据被回显到页面中的注入,即执行 SQL 查询,若其报错信息能回显到页面中,那么可直接进行有回显的 SQL 注入。在回显注入中,攻击者利用查询语句返回的报错信息可以判断数据库中的字段数、显示位,查看表名、列名,还可以利用函数获得其他更多的信息。盲注是指在 SQL 注入过程中,由于服务器并不将攻击者想要知道的报错信息回显到前端页面,攻击者需要在不知道数据库返回值的情况下,利用一些方法对数据中的内容进行猜测或者尝试,以实施 SQL 注入。盲注一般可分为两类。

1) 布尔盲注

基于布尔型 SQL 盲注简称布尔盲注,即在 SQL 注入过程中,Web 页面仅返回 True 和 False,这时无法通过根据页面返回的信息得到所需的数据库中的相关信息,但是可以通过构造逻辑判断(比较大小)来获取想要的信息。

2) 时间盲注

注入了 SQL 代码之后,存在以下两种情况:

(1) 如果注入的 SQL 代码不影响后台数据库的正常功能,那么 Web 应用的页面显示正确(原始页面)。

(2) 如果注入的 SQL 代码影响后台数据库的正常功能,即产生了 SQL 注入,然而由于 Web 应用程序采取了重定向或屏蔽措施,此时 Web 应用的页面仍然会显示正常。

这时,无法返回页面信息判断注入的 SQL 代码是否已被后台数据库执行,即无法判断 Web 应用程序是否存在 SQL 注入。

因为基于布尔型 SQL 盲注的前提是 Web 程序返回的页面存在 True 和 False 两种不同的页面信息,所以在这种情况下,基于布尔型 SQL 盲注很难发挥作用。对这种情况,采用基于 Web 应用响应时间上的差异来判断是否存在 SQL 注入,即基于时间型 SQL 盲

注(简称时间盲注),在加入特定的时间函数后,通过查看 Web 页面返回的时间差来判断注入的语句是否正确。

2. SQL 注入攻击的流程

SQL 注入攻击分为一阶 SQL 注入攻击和二阶 SQL 注入攻击。一阶 SQL 注入攻击是指注入攻击发生在一次 HTTP 数据请求和响应中,其注入漏洞利用的流程如下:

(1) 攻击者通过构造数据的形式,在浏览器或者其他软件中提交 HTTP 数据报文请求到服务器端进行处理,提交的数据报文请求中可能包含了攻击者构造的恶意 SQL 语句或者命令信息。

(2) 服务器端应用程序将对攻击者提交的 HTTP 数据报文请求进行处理,则攻击者构造的 SQL 注入语句或命令将在服务器端环境中执行。

(3) 服务器端会返回执行的结果数据信息,黑客可以通过执行的结果数据信息判断注入漏洞利用是否成功。

二阶 SQL 注入攻击与一阶 SQL 注入攻击不同。相对于一阶 SQL 注入攻击流程而言,二阶 SQL 注入攻击的流程一般如下:

(1) 攻击者通过构造数据的形式,在浏览器或者其他软件中提交 HTTP 数据报文请求到服务器端进行处理,提交的数据报文请求中包含了攻击者构造的恶意 SQL 语句或者命令。

(2) 服务器端应用程序会将攻击者提交的数据信息存储起来,通常是保存在数据库中,保存这些数据信息的主要作用是为应用程序执行其他功能提供原始输入数据,并对客户端请求做出响应。

(3) 攻击者第二次向服务器端发送一个与第一次不同的请求数据信息。

(4) 服务端接收到攻击者提交的第二个请求信息后,为了处理该请求,服务器端会查询数据库中已经存储的数据信息并进行处理,从而导致攻击者在第一次请求中恶意构造的 SQL 语句或者命令在服务端环境中执行。

(5) 服务器端返回执行的处理结果数据信息,黑客可以通过返回的结果数据信息判断二次注入漏洞利用是否成功。

一阶 SQL 注入攻击和二阶 SQL 注入攻击危害是一样的,攻击者获得数据库的访问权限,从而窃取相关数据,但是一阶 SQL 注入攻击可以通过相关工具扫描出来,而二阶 SQL 注入攻击是一种更加细微的漏洞,通常很难被检测。从二阶 SQL 注入漏洞利用的流程可以发现,二阶 SQL 注入攻击对漏洞的利用相对于一阶 SQL 注入攻击来说不仅仅是多发送一次请求这么简单,它需要攻击者对应用程序的功能有完整的理解。也就是说,要想实现二阶 SQL 注入漏洞的利用,要求攻击者对不同功能之间的关系有一定的理解。通常二阶 SQL 注入漏洞的测试主要依据测试人员对系统功能的理解和对常出错位置的经验判断。但是随着应用功能的增加,经验性的测试并不能保证测试结果。

3. SQL 注入攻击的防范措施

SQL 注入攻击能使攻击者绕过认证机制,控制远程服务器上的数据库,从数据库中

获取敏感信息，在数据库中添加数据库操作用户，从数据库中导出文件，甚至获取数据库系统的管理员权限，危害范围广且性质严重，因而防范 SQL 注入势在必行。几种常见的 SQL 注入防御措施如下：

1）参数化语句

参数化语句是一种安全的动态查询创建方式，可以避免一般的 SQL 注入漏洞。在一般情况下，可以用来代替动态查询。由于现在的数据库具有查询优化能力，因此这种方法具有很高的查询速度和执行效率。Java 禁止通过字符串连接的方法直接使用用户输入构造可执行 SQL 语句，但允许编程人员通过参数化语句，使用占位符或者绑定的变量来创建 SQL 查询语句，以替代直接使用用户的输入来创建 SQL 查询语句。

对于 Java 语言，全面使用参数化执行语句即通过使用预编译语句代替直接的语句执行，类型化 SQL 参数通过检查输入的类型，确保输入值在数据库中当作字符串、数字、日期或布尔值等而不是可执行代码进行处理，从而防止 SQL 注入攻击。此外，对于 Java 数据库连接 JDBC 而言，SQL 注入攻击只对语句有效，对预编译语句是无效的，这是因为预编译语句不允许在不同的插入时间改变查询的逻辑结构。例如，验证用户是否存在的 SQL 语句为

```
用户名'and pswd='密码
```

如果在用户名字段中输入

```
'or 1=1
```

或在密码字段中输入

```
'or 1=1
```

将绕过验证，但这种手段只对语句有效，对预编译语句无效。

使用预编译语句防范 SQL 注入攻击的优点是：多次运行速度快；防止数据库缓冲区溢出；代码的可读性和可维护性好。这些优点使这种方法成为访问数据库的语句对象的首选。不过其缺点也很明显，由于灵活性不够好，在有些场合必须使用语句。

2）输入验证

输入验证就是验证用户的输入是否符合系统定义的标准，它可能简单到直接验证一个参数的类型，也可能复杂到使用正则表达式或者业务逻辑去验证用户输入。如果构造 SQL 指令时需要动态加入约束条件，可以通过创建一份合法字符串列表，使其对应于可能要加入 SQL 指令中的不同元素来避免 SQL 注入攻击。在 Java 中，主要对输入参数的长度、范围和类型进行校验。

（1）校验输入数据的长度。如果输入数据是字符串，必须校验字符串的长度是否符合要求，进行长度校验会加大攻击者实施攻击的难度。

（2）校验输入数据的范围。如果输入数据是数值，必须校验数据的范围是否正确，例如年龄应该为 0～150 的正整数。

（3）校验输入数据的类型。如果仅允许输入的数据是数字，那么就不应该接收字符型的数据。

3）纵深防御

采用纵深防御措施的好处是：一旦前线防御失效，可以提供额外的保护。在攻击后端数据库的上下文过程中，有 3 个层次的纵深防御措施需要实施。

(1) 应用程序在访问数据库时应使用尽可能低级别的特权。

通常应用程序并不需要数据库管理员(Database Administrator，DBA)级别的权限，只需要读取和写入数据。在不同情况下，应用程序会使用不同的数据库账户进行不同的操作。例如，如果 90%的数据库查询只需要读权限，则这些操作可以用一个没有写权限的账户来执行；如果某个特定的查询只需要读取一个子集的数据(例如命令表，而不是用户账户表)，则使用一个具有相应访问权限的账户即可。如果在整个应用程序中强制实行这种做法，则 SQL 注入漏洞的影响将大大减小。

(2) 数据库中要去掉所有的无用的功能。

将数据库中所有不必要的功能删除或禁用。尽管在某些情况下，熟练的攻击者可以通过其他手段生成一些新功能，但是数据库的小范围冻结仍然可以成为攻击者的障碍。

(3) 数据库补丁的更新要进行评估且及时更新。

数据库本身的漏洞可能成为攻击者利用的对象。补丁要及时更新，但是对补丁要进行评估，因为一些不必要的补丁或插件也许更方便攻击者实施攻击。

4）输出编码

除了对用户输入进行验证外，还需要对程序各个模块之间或者各个部分之间传递的数据进行编码。在可能存在 SQL 注入漏洞的环境中，为了保证传递给数据库的数据不会被错误地处理，则需要对输出进行编码。

有一种情况经常会被忽略，就是当编码的信息来自数据库中的数据，尤其是使用过的数据时，通常不会对这些数据进行验证或者过滤处理。在这种情况下，虽然不会直接导致 SQL 注入漏洞，但还是应当考虑采用类似的编码方式进行处理，以防止产生其他安全漏洞，如 XSS 漏洞等。

4.2.2 XSS 漏洞

跨站脚本攻击(Cross Site Scripting，XSS)是一种针对客户端浏览器的注入攻击。与 SQL 注入攻击不同的是，XSS 攻击者将恶意脚本注入 Web 应用程序中并不是为了攻击应用程序本身，而是将 Web 应用程序作为攻击其他网站的中转站。当其他用户访问被注入恶意脚本的 Web 应用程序时，恶意脚本就会被下载到该用户的浏览器中并运行。被注入的恶意代码能够在支持 HTML、JavaScript、Flash、ActiveX、VBScript 等功能的客户端浏览器上执行，造成主机上的敏感信息泄露、Cookie 被窃取、配置被更改、重定向到其他网站等后果。

XSS 漏洞产生的主要原因有两个：一是由于 HTML 协议无法区分数据和代码，即无法明确指出用户输入的非法数据，则攻击者可以将恶意代码注入 HTML 代码中；二是 Web 应用程序将用户数据发送回浏览器时没有做适当的转义处理，使得包含恶意脚本的数据被放入 Web 网页中，在客户端浏览器中执行，从而引发 XSS 漏洞。

1. XSS 漏洞的分类

XSS 漏洞根据注入位置和触发流程的不同分为 3 类，分别是反射型 XSS 漏洞、存储型 XSS 漏洞和基于 DOM 型 XSS 漏洞。

1）反射型 XSS 漏洞

反射型 XSS 漏洞也称为永久型 XSS 漏洞，是目前最流行的一种 XSS 漏洞。典型的反射型 XSS 漏洞攻击是攻击者将攻击代码（例如 URL 中的数据、HTTP 协议头的数据和 HTML 表单中提交的数据）存储在客户端上，而不是存储在服务器上。应用程序通过 Web 请求获取不可信赖的数据，在未检验数据是否存在恶意代码的情况下，便将其传送给用户，在用户浏览器上执行攻击代码，达到攻击者窃取用户的键盘记录、窃取用户的 Cookie、窃取剪贴板内容、篡改网页内容等目的。

反射型 XSS 漏洞攻击的过程如图 4-1 所示。

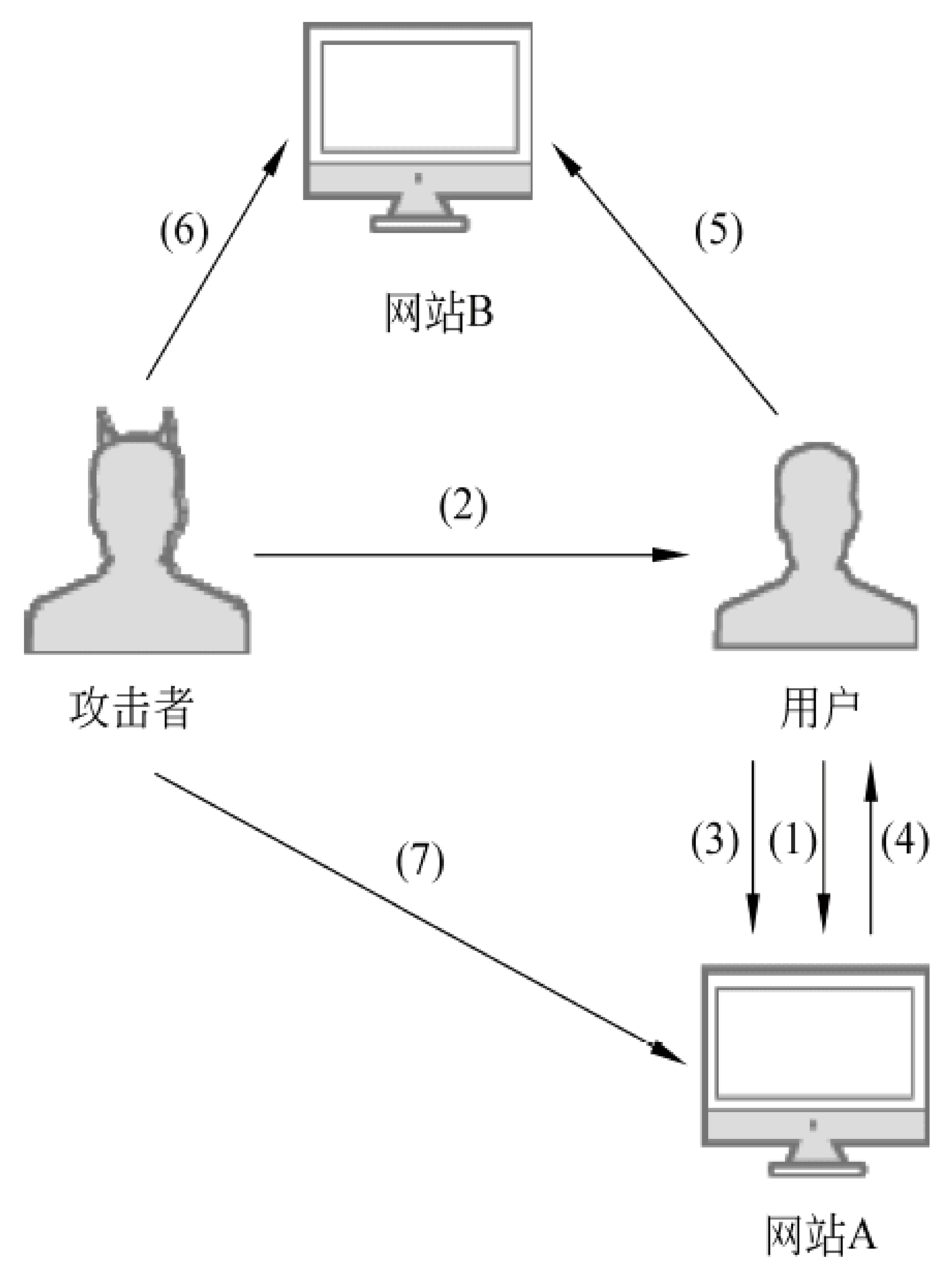


图 4-1　反射型 XSS 漏洞攻击过程

（1）用户有访问网站 A 的权限，登录成功后将得到一个会话 Cookie。

（2）攻击者发现网站 A 存在反射型 XSS 漏洞，于是构造了含有攻击脚本的恶意 URL，并诱使用户点击这个 URL 来访问网站 A。

（3）用户点击了攻击者提供的 URL，登录到网站 A 上。

（4）网站 A 根据 URL 中的参数将恶意 JavaScript 代码插入网页中，返回给用户。

（5）恶意代码在用户的浏览器上执行，将用户的会话 Cookie 发送给攻击者控制的网站 B。

（6）攻击者从网站 B 得到用户的会话 Cookie。

（7）攻击者劫持了用户会话，实施攻击。

反射型 XSS 漏洞攻击和钓鱼攻击非常相似，都是诱使被攻击者访问某个恶意的

URL,从而获取用户敏感信息。但钓鱼攻击是用户点击 URL 后进入一个伪造的网页,该网页和用户想要访问的 Web 应用程序不在同一个域中,则根据同源策略,该钓鱼网页无法直接获得用户在另一个域的 Web 应用程序中的 Cookie 等敏感信息,所以钓鱼网页还需要诱导用户输入用户名和口令等信息,钓鱼攻击才能成功。而在反射型 XSS 漏洞攻击中,用户访问的恶意 URL 位于用户原本想要访问的 Web 应用程序所在的域中,并不违反同源策略,即恶意的 URL 可以自动执行 JavaScript 代码来窃取用户的敏感信息。这些操作不需要用户做进一步的配合,就可以自动完成,即当用户访问了含有反射型 XSS 漏洞的 URL 时,XSS 漏洞攻击就已经成功了。

2) 存储型 XSS 漏洞

存储型 XSS 漏洞是指攻击的恶意脚本被存储在服务器端的数据库或者文件中,在访问服务时,应用程序从数据库或其他后端数据存储中获取了不可信赖的数据,在未检验数据是否存在恶意代码的情况下,便将其传送给 Web 用户,这样就会导致存储型 XSS 攻击。其实现原理如下：XSS 恶意脚本提交至服务器端→服务器端将恶意脚本存入数据库→当服务再次被请求时,服务器端回显被植入恶意脚本的数据给客户端→客户端执行恶意脚本,形成攻击。

存储型 XSS 漏洞攻击的过程如图 4-2 所示。

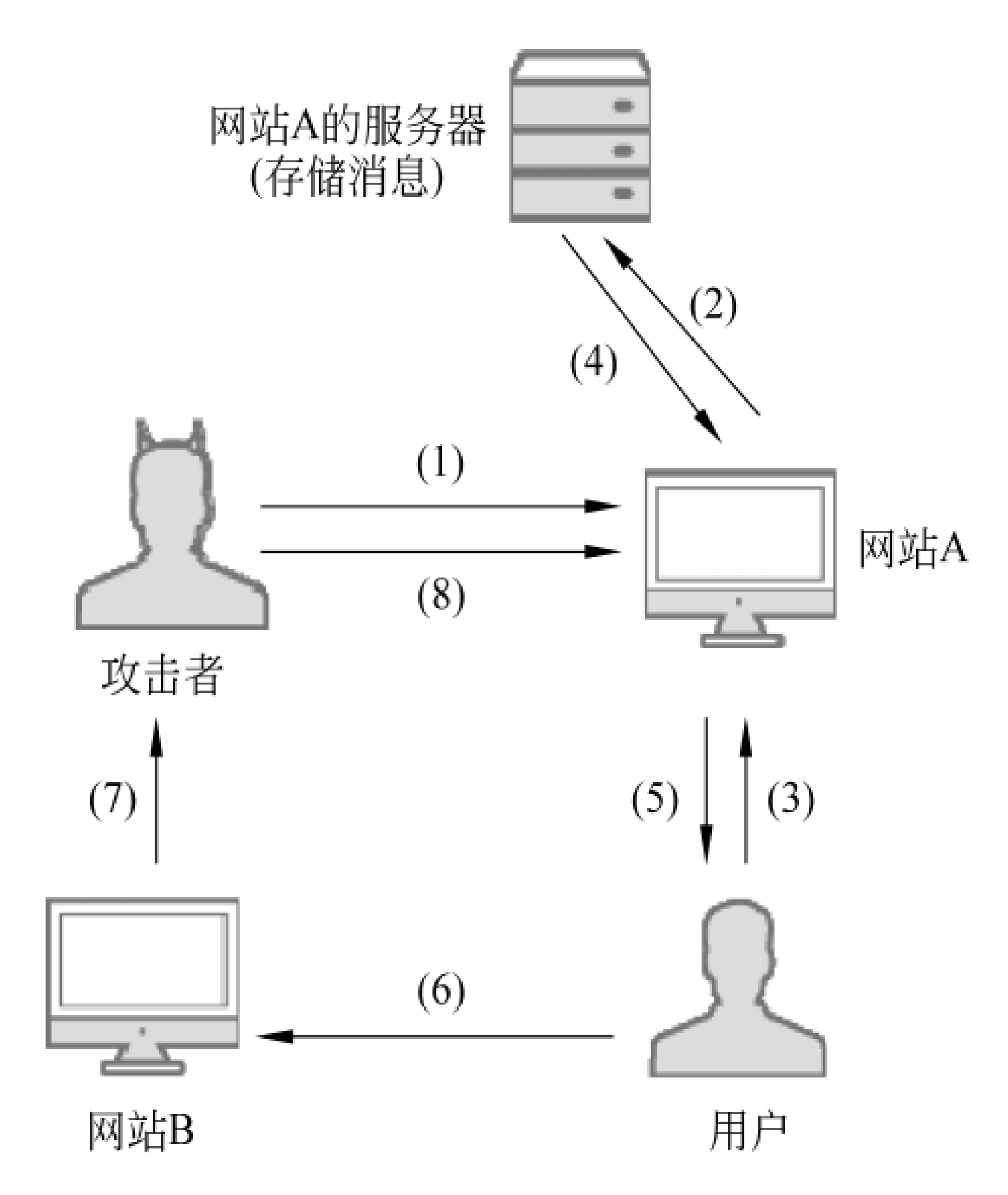


图 4-2　存储型 XSS 漏洞攻击过程

(1) 攻击者发现在网站 A 上存在存储型 XSS 漏洞,于是发送一个恶意消息到网站 A 上。

(2) 网站 A 存储此恶意消息。

(3) 用户发现该消息,点击链接进行读取。

(4) 网站 A 将存储的恶意消息取出。

(5) 用户读取了该恶意消息,恶意代码在用户的浏览器上执行。

(6) 恶意代码将用户的 Cookie 发送到网站 B 上。

(7) 攻击者从网站 B 得到用户的 Cookie。

(8) 攻击者劫持了用户的会话，访问网站 A。

3) DOM 型 XSS 漏洞

DOM 型 XSS 漏洞又称作本地 XSS 漏洞，此类型的漏洞存在于页面中的客户端脚本中。当页面中的 JavaScript 代码访问了 URL 请求参数，并且未经编码便直接使用相应的参数信息在页面中输出某些 HTML 时，就有可能出现此类型的 XSS 漏洞。

DOM 型 XSS 漏洞攻击过程如图 4-3 所示。

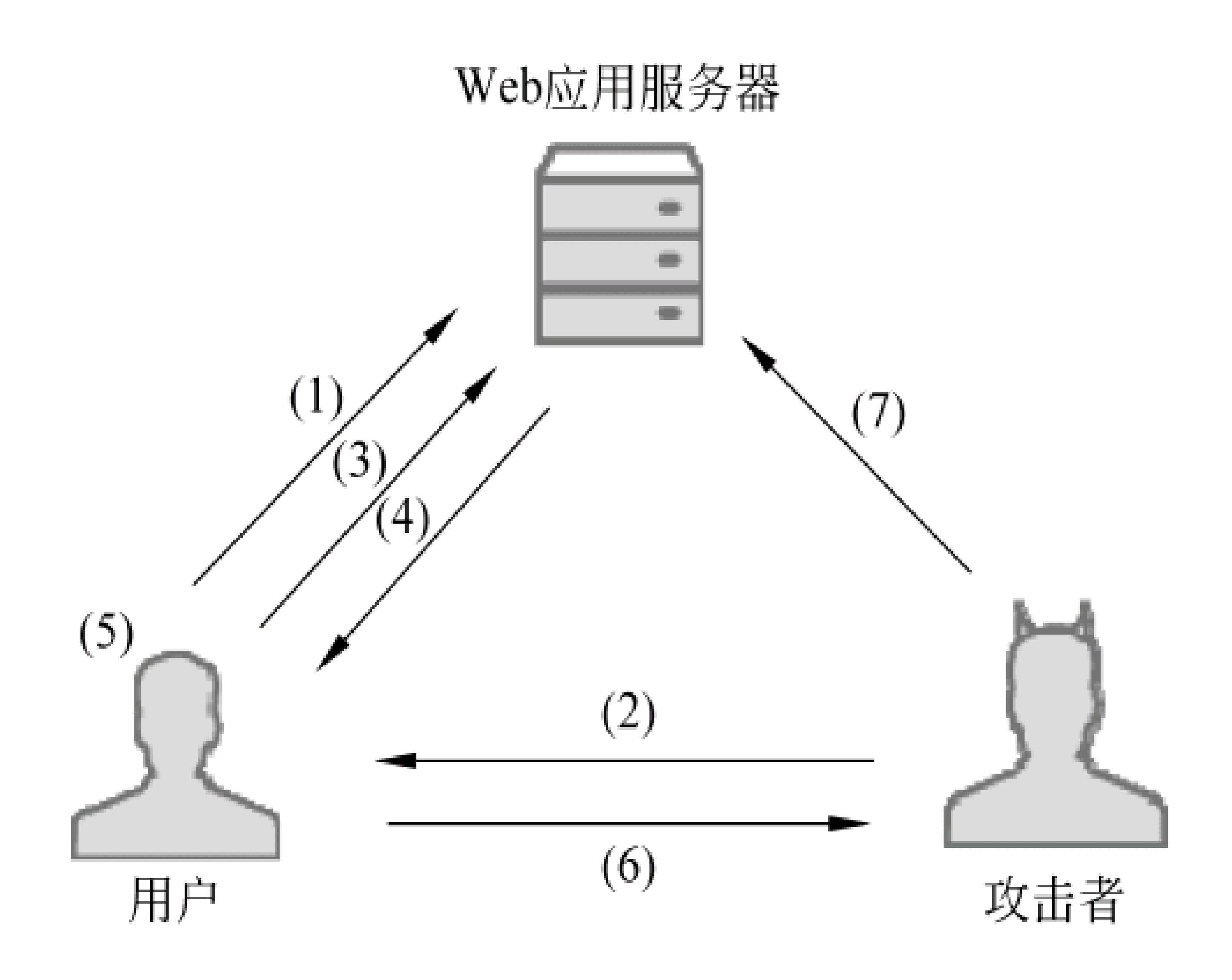


图 4-3　DOM 型 XSS 漏洞攻击过程

(1) 用户登录 Web 应用。

(2) 攻击者发给用户一个 URL。

(3) 用户点击攻击者发来的 URL。

(4) 服务器返回包含 JavaScript 脚本的页面。

(5) 用户浏览器把页面的 HTML 文本通过 DOM 解析，浏览页面内容。

(6) 恶意脚本在 DOM 解析时开始执行，传送用户敏感信息给攻击者。

(7) 攻击者对 Web 应用发动攻击。

对于 DOM 型 XSS 漏洞，由于所有的 URL 处理方式对用户都是可见的，攻击者可以轻易发现网页中的 XSS 漏洞并实施攻击。而对于前面两类 XSS 漏洞，用户对 Web 服务器所处理的数据是不可知的，因此攻击者发现 XSS 漏洞要困难一些。

2 XSS 攻击的防范措施

由于 XSS 攻击可以使用多种客户端语言来实现，也可以跨越多种操作系统平台，利用方式多样化，并具有较强的隐蔽性，因此近年来其发生频率大幅上升，危害性也越来越大。攻击者可以利用 XSS 漏洞窃取用户敏感信息，重写 Web 网页，窃取用户会话，将用户重定向到钓鱼网站，甚至结合其他手段进一步控制用户的系统。因此，了解一些防范 XSS 攻击的技术越来越重要。下面详细描述了 XSS 攻击的几种常见防范方式。

XSS 漏洞采用两种方法来防范：一种方法是增强 Web 应用程序本身的安全性；另一种方法是增强 Web 应用程序运行平台的安全性，包括部署 Web 应用级防火墙、设置数据库安全措施以及其他的防护措施等。下面主要介绍增强 Web 应用程序本身的安全性的

方法。

增强 Web 应用程序本身的安全性就是在设计应用程序时充分考虑如何避免 XSS 漏洞的问题。通常采用 HttpOnly、输入验证、输出编码、规范化等方法来增强 Web 应用程序本身的安全性。

1）HttpOnly

HttpOnly 最早是由微软公司提出，并在 IE 6 中实现的，至今已成为一种标准。HttpOnly 防范的是 XSS 攻击后的 Cookie 劫持攻击。Cookie 攻击的过程如下：

（1）浏览器向服务器发起请求，这时没有产生 Cookie。

（2）服务器返回时发送 Set-Cookie 头，向客户端浏览器写入 Cookie。

（3）在该 Cookie 到期前，浏览器访问该域下的所有页面时，都将发送该 Cookie。

HttpOnly 是在第二步发送 Set-Cookie 时标记的。通过在浏览器设置中禁止页面的 JavaScript 代码访问带有 HttpOnly 属性的 Cookie 来防范 XSS 攻击。

2）输入验证

在进行输入验证时，可以采用白名单验证和黑名单验证两种方式。对于白名单验证，根据白名单对 URL、查询关键字、HTTP 头、POST 数据等进行检查，只接收指定长度范围内、采用适当格式和预期字符的输入，对其他内容的输入一律过滤掉。对于黑名单验证，根据黑名单对包含 XSS 代码特征的内容进行过滤，如<、>、script、javascript 等内容。白名单验证和黑名单验证都采用正则表达式进行检查和验证。然而输入验证很难过滤掉脚本中所有的非法字符，因为所有的字符在 HTML 字符集里都是合法的，因此输入验证方法有一定的局限性。

3）输出编码

对所有输出字符进行 HTML 编码，把包括 HTML 标记在内的危险字符转换成无害的 HTML 编码表示，将用户输入的 HTML 脚本当成普通文字来处理，而不会成为目标页面 HTML 部分执行的代码，例如，字符串“<script>”可以编码为“< script>”等，这样浏览器便对它进行转义和逐字解析。然而输出编码方法有可能影响 Web 应用的交互性，因为网页不接受由用户输入的任何 HTML 内容，如博客和社交论坛里由用户提交的 HTML。

4）规范化

通过规范化，将输入变为规范格式或者简单格式，使之只包含最小的、安全的 Tag，即不包含 JavaScript；去掉对远程内容的引用，尤其是样式表和 JavaScript；使用 Cookie 时应设置 HTTP Only 属性；等等。

4.2.3　重定向漏洞

随着网络技术的发展，应用程序越来越多地需要和第三方应用进行交互，以及在自身应用内部根据不同的逻辑将用户引导到不同的页面。例如，一个典型的登录接口就经常需要在认证成功之后将用户引导到登录之前的页面。在整个过程中，如果验证措施不够完善，就可能导致一些安全问题，在特定条件下可能引起严重的安全漏洞，重定向就是其中一种。

在 Java 中，实现 Web 页面跳转有两种方式：一种是转发，另一种是重定向。转发是服务器请求资源并直接访问目标地址的 URL，读取 URL 的响应内容后，再把读取的内容发送给浏览器。浏览器并不知道服务器发送的内容是从哪里获取的，因为这个跳转过程是在服务器端而不是客户端实现的，所以客户端并不知道这个跳转动作，即客户端的地址栏中还是原来的地址。重定向则是服务器端根据逻辑，发送一个状态码给客户端，通知客户端浏览器请求新的地址进行访问，所以此时客户端地址栏将显示新的地址。例如，当用户访问 http://www.sun.com 时，发现浏览器地址栏中的 URL 会变成 http://www.oracle.com/us/sun/index.htm，这就是重定向。一般来说，转发比重定向快，重定向是在客户端完成的，转发是在服务器端完成的。

1. 重定向的原理

在 Web 应用中，重定向是非常普遍的，例如一些知名公司会买下与自己的网站域名相似的域名，以便用户在访问时即使不小心拼错域名也可以成功访问正确的网站。然而，如果应用程序允许将未验证的输入数据传递给 HTTP 重定向函数，攻击者便可通过修改输入数据发动攻击，引导用户访问他们所构造的恶意网站。根据 OWASP，开放重定向漏洞出现在应用程序接收输入参数并将用户重定向到带有该参数值的新地址，并且没有对输入值进行任何校验的时候。攻击者常利用此漏洞进行钓鱼攻击，滥用用户对于原网站的信任，并将恶意网站伪装成合法网站的样子作为重定向目的地，从而让用户在无意中浏览恶意网站。

重定向攻击造成的危害很大，攻击者利用重定向漏洞欺骗安全意识低的用户，例如引导用户进入"中奖"页面，窃取用户身份认证凭据，收集用户的个人敏感信息，给用户造成经济损失。另外，对于一些包含在线业务的企业，如淘宝等，重定向攻击也会严重影响企业声誉，影响业务正常运营。

在 Java 中，开发人员通常通过 HTTP Servlet Response 对象的 sendRedirect()函数将用户自动重定向到另一个页面，该方法相当于浏览器重新发送一个访问请求：

```
response.sendRedirect(path);
```

其中，URL 需要在代码中被明确声明，例如：

```
response.sendRedirect("http://www.mysite.com");
```

此条指令明确声明将页面重定向到 http://www.mysite.com，从而使攻击者无法对地址进行更改，防止重定向攻击。

如果不使用输入验证或其他方法来验证 URL 的真实可靠性，攻击者就可以将用户重定向到恶意网站，对用户进行钓鱼诈骗。攻击者还可以创建超链接，将用户重定向到未经过验证的与原始地址十分相似的恶意网站，例如：

```
http://example.com/example.php?url=http://malicious.example.com
```

用户看到指向原始受信任站点(example.com)的链接后，将很难意识到已经发生的重定向。

此外，当应用程序允许利用用户输入在网站的不同部分之间转发请求时，应用程序必须检查用户是否有权访问 URL 以执行它提供的功能。如果应用程序未能执行这些检查，则攻击者恶意构造的 URL 可能会通过应用程序的访问控制检查，从而将攻击者转发到通常不允许的管理功能。

2 重定向的防范措施

重定向与跨站请求伪造攻击的防御方式类似，开发人员需要对传入的 URL 进行有效性认证，以确保该 URL 来自可靠的地址。通常人们通过创建可信 URL 列表（主机列表或正则表达式）来检查输入，然而这同样存在着被绕过的安全风险。例如，一些应用允许引入可信网站（如 youku. com）的视频，其限制方式往往是检查 URL 是否是由 youku. com 来实现的，如果在 youku. com 内存在一个 URL 跳转漏洞，将导致最终引入的资源属于不可信的第三方资源或者恶意网站，从而产生安全问题。下面描述两种常见的重定向防范措施。

1）referer 限制

referer 限制能够检查访问请求是否来自合法的服务器。如果能确定传递 URL 参数的输入的来源，开发人员可以通过该方式实现安全限制，保证 URL 的有效性，避免恶意用户自己生成跳转链接。该方法的一个明显的缺陷是，不是所有的服务器端都可以得到 referer 信息，因为很多浏览器出于信息安全考虑会阻止发送 referer 信息。

2）加入有效性验证标记

通过在生成的链接内加入用户不可控的有效性验证标记对生成的链接进行校验，可以避免用户生成自己的恶意链接从而被利用。

4.2.4　路径遍历漏洞

在 Web 应用中，当允许外界以参数的形式来指定服务器上的文件名时，如果没有对文件名进行充分校验，就可能会导致目录遍历漏洞。路径遍历漏洞允许攻击者突破 Web 应用程序的安全控制，访问受限的目录，获取系统文件及服务器的配置文件，如日志、源代码等敏感数据，甚至采取更危险的行为，在 Web 服务器的根目录以外执行造成系统崩溃的指令。

1. 路径遍历的原理

大多数 Web 应用程序都有读取查看服务器文件的功能，用提交的参数来指明文件名，例如，http://www. nuanyue. com/getfile＝image. jpg，当服务器处理传送过来的 image. jpg 文件名后，Web 应用程序会自动为其添加完整路径，形如 d://site/images/image. jpg，将读取的内容返回给访问者。从表面来看，这只是文件交互的一种简单的过程，但是由于文件名可以任意更改，而且服务器支持“～/”“../”等特殊符号的目录回溯，从而使攻击者可以越权访问或者覆盖系统敏感数据，如网站的配置文件、系统的核心文件等，这样的缺陷被命名为路径遍历漏洞。

Web 服务器主要提供两个级别的安全机制：访问控制列表（Access Control List，ACL）和根目录访问。访问控制列表是用于授权过程的，Web 服务器的管理员利用 ACL

限制特定的用户或用户组在服务器上访问、修改和执行某些文件的权限。根目录是服务器文件系统中的一个特定目录，对用户来说它就像一个限制，使用户无法访问位于这个目录以上的任何内容。例如，在 Windows 的 IIS 服务器中，其默认的根目录是 C：\Inetpub\wwwroot，则用户一旦通过了 ACL 的检查，就可以访问 C：\Inetpub\wwwroot 目录以及其他所有位于这个根目录以下的目录和文件，但用户无法访问 C：\Windows 目录。根目录的存在能够防止用户访问服务器上的一些关键性文件，例如 Windows 平台上的 cmd. exe 或 Linux/UNIX 平台上的口令文件。

路径遍历漏洞形成的主要原因是 Web 网站对 Web 内容缺乏恰当的访问控制，攻击者通常通过访问根目录，发送一系列的. . /字符遍历目录，或通过根目录访问 Web 应用所在目录之外的路径，以遍历更高层的目录。路径遍历漏洞十分容易发现，只需对 Web 应用程序的读写功能块进行手工检测，就能通过返回的页面内容直观地判断是否存在路径遍历漏洞。攻击者只需要一个 Web 浏览器和关于系统的一些默认文件和目录所在的位置的知识，即可发动路径遍历攻击。

2 路径遍历的攻击场景

路径遍历漏洞可能存在于 Web 服务器软件本身，也可能存在于 Web 应用程序的代码之中。

1）利用 Web 应用代码进行路径遍历攻击的应用场景

在包含动态页面的 Web 应用中，输入往往是通过 GET 或 POST 的请求方法从浏览器获得的。下面是一个 GET 的 HTTP URL 请求示例：

```
http://test.webarticles.com/show.asp?view=oldarchive.html
```

浏览器用这个 URL 向服务器发送了对动态页面 show. asp 的访问请求，并且同时发送了值为 oldarchive. html 的 view 参数，当访问请求在 Web 服务器端执行时，show. asp 会从服务器的文件系统中取得 oldarchive. html 文件，并将其返回给客户端的浏览器，此时攻击者就可以假定 show. asp 能够从文件系统中获取文件，于是恶意构造如下的 URL，希望 show. asp 能够从文件系统中获取 system. ini 文件并返回给用户：

```
http://test.webarticles.com/show.asp?view=../../../../../Windows/system.ini
```

虽然攻击者需要耗费精力去猜测需要往上遍历多少层才能找到 Windows 目录，但这其实并不困难，经过若干次的尝试便可成功。

2）利用 Web 服务器进行路径遍历攻击的实例

除了 Web 应用的代码以外，Web 服务器本身也可能存在路径遍历漏洞，这些漏洞可能存在于 Web 服务器软件或一些存放在服务器上的示例脚本中。虽然在最近的 Web 服务器软件中这个问题已经得到解决，但是网上的很多 Web 服务器仍然使用老版本的 IIS 和 Apache，而它们则可能仍然无法抵御这类攻击。另外，即使用户使用的是已经解决了这个问题的 Web 服务器软件版本，仍然可能存在一些存有敏感默认脚本的目录。

3. 路径遍历漏洞的检测

路径遍历漏洞一般隐藏在文件读取或者展示图片的功能块中，一般采用通过参数提交上来的文件名的形式，从中可以看出，过滤交互数据是完全有必要的。攻击者利用对文件的读取权限进行跨越目录访问，例如，通过"../../../../../../../etc/passwd"或者"../../../../boot.ini"等形式访问一些受控制的文件。当然现在部分网站有类似WAF的防护设备，只要在数据中有/etc /boot.ini 等文件名出现，网站就会直接进行拦截。路径遍历漏洞其实并不复杂，主要是对 Web 应用程序的文件读取交互的功能块进行检测。以下两种方法都可以检测出该漏洞：

(1) 利用 Web 漏洞扫描器对应用程序进行扫描。Web 漏洞扫描器通过遍历用户 Web 站点的所有目录来判断是否存在路径遍历漏洞，如果有，Web 漏洞扫描器会报告该漏洞并给出对应的解决方法。除了目录遍历漏洞以外，Web 漏洞扫描器还能检查 SQL 注入、跨站点脚本攻击以及其他常见的安全漏洞。

(2) 查看 Web 日志中是否有未授权用户访问越级目录的事件，如果有，则说明存在路径遍历漏洞。Web 应用服务器日志中会记录针对用户的访问行为，如果发现未授权用户访问越级目录，或者通过 url 参数对规定目录之外的系统进行操作，基本可以判定路径遍历攻击行为正在发生。

4. 路径遍历攻击的防范

防范路径遍历攻击最有效的办法就是权限控制，即谨慎地处理传向文件系统 API 的参数。由于大多数目录或者文件权限都没有得到合理的配置，而 Web 应用程序对文件的读取大多依赖于系统本身的 API，在参数传递的过程中，如果系统的控制检测不够严格，就会出现越权现象。

以下是几种防范路径遍历攻击的方法，根据不同情况可以组合使用：

(1) Web 应用程序可以使用 chrooted 环境访问包含被访问文件的目录，或者使用"绝对路径+参数"来访问文件目录，使其即使是越权或者跨越目录访问，也只能被限制在指定的目录下。常用的 www 目录就是一个 chroot 应用。chroot 是 UNIX 系统的一个操作，用来改变程序执行时所参考的根目录位置。一个运行在此环境下，经由 chroot 设置根目录的程序，将无法对指定根目录之外的文件进行访问，读取或更改文件内容。由 chroot 设置的根目录被形象地称为"chroot 监狱"(chroot jail 或 chroot prison)。

(2) 数据净化。对网站用户提交的文件名参数进行硬编码或者统一编码，对文件类型利用白名单进行控制，对包含恶意符号或者空字节的输入进行过滤。

(3) 由于有时攻击者会仿冒网站的其他用户来执行操作，此时就需要对 Web 网站的用户进行授权设置。全面进行 Web 站点安全设置，防止攻击者能够仿冒网站的其他用户来执行操作；确保 Web 网站对用户的安全分级授权方式能禁止其越级和跨出合法区域的访问。对使用的 Web 应用系统进行升级和安装防漏洞补丁。

(4) 注意文件名的使用规范。固定文件名；避免由外部指定文件名；使用编号代替文件名；文件名不允许包含目录名；文件名中不能包含/、\、：等在目录中使用的字符；限定文件名中仅包含字母和数字。

4.2.5 不安全的安全哈希算法

哈希(hash)的意思为散列。在哈希算法中,输入任意长度的字符串,算法能给出固定长度的字符串输出,输出的字符串一般称为哈希值,典型的哈希算法包括 MD4、MD5 和 SHA-1 等。

安全哈希算法(Secure Hash Algorithm,SHA)是经 FIPS(Federal Information Processing Standards,联邦信息处理标准)认证的算法,能计算出一个数字消息所对应的长度固定的字符串(又称消息摘要)。SHA 家族包括 5 个算法,分别是 SHA-1、SHA-224、SHA-256、SHA-384 和 SHA-512。其中 SHA-1 在许多安全协定中广为使用,曾被视为 MD5(更早之前被广为使用的哈希函数)的后继者。但是,近些年 SHA-1 的安全性受到密码学家的严重质疑。

SHA-1 主要适用于数字签名标准(Digital Signature Standard,DSS)中定义的数字签名算法(Digital Signature Algorithm, DSA)。对于长度小于 2^{64} b 的消息,SHA-1 会产生一个 160 位的消息摘要。当用户接收到消息的时候,包含在这个消息中的消息摘要可以用来验证数据的完整性。在传输的过程中,数据很可能会发生变化,这时候就会产生不同的消息摘要。SHA-1 有如下特性:不能从消息摘要中复原信息;两个不同的消息不会产生同样的消息摘要(其实会有 10^{-48} 的概率出现相同的消息摘要,不过一般在实际应用时忽略这一点)。

1. 安全哈希算法的原理和用途

哈希算法之所以被认为是安全的,主要是基于以下两种性质:

(1) 无冲突。无法找到得到相同输出值的两个不同输入。也就是当知道 x 时,无法求出一个 y,使 x 与 y 的哈希值相同。无冲突使得哈希算法对原始输入的任意一点进行更改,都会导致产生不同的哈希值,因此哈希算法可以用来检验数据的完整性。用户在一些网站下载某个文件时,网站会提供此文件的哈希值,以供用户下载文件后检验文件是否被篡改。

(2) 不可逆。不可从结果推导出它的初始状态。也就是当知道 x 的哈希值时,无法求出 x。不可逆的特性使哈希算法成为一种单向密码体制,只能加密,不能解密,常用来加密用户的登录密码等凭证。

然而,由于不是一对一的映射,哈希函数转换后不可逆,即不可能通过逆操作和哈希值还原出原始的值,受到计算能力的限制,也无法还原出所有可能的全部原始值。而且由于哈希函数的值域有限,理论上会有无穷多个不同的原始值,它们的哈希值都相同。MD5 和 SHA 能做到的是求逆和求碰撞在计算上不可能,也就是正向计算很容易,而反向计算即使穷尽人类所有的计算资源都做不到。

哈希算法的用途很多,主要有以下 3 种:

(1) 哈希算法运用在字典表等需要快速查找的数据结构中,其计算复杂度很低,不会随着数据量增加而增加。在文件签名中,将文件内容通过哈希算法处理后得到一个哈希值。为了验证这个文件是否被修改过,只需要把文件内容用同样的哈希算法处理,之后对

比计算得到的哈希值和随文件一起传送过来的哈希值即可。如果不公开哈希算法,那么信道是无法在篡改文件内容的时候篡改文件哈希值的。不过一般在应用的时候,哈希算法是公开的,这时候会用一个非对称加密算法加密这个哈希值,这样即使能够计算出文件的哈希值,但由于攻击者没有加密密钥,依然无法篡改加密后的哈希值。哈希算法也可以用在电子签名中。

(2) 哈希算法可用来进行破解,这种破解不是传统意义上的解密,而是按照已有的哈希值构造出能够计算出相同哈希值的其他原文,从而干扰原文的不可篡改性的验证,俗称“碰撞”。这种碰撞对现有的电子签名危害并不严重,因为解密时需要构造出有意义的原文,否则只是构造了一个完全不可识别的原文,要么接收系统无法处理而报错,要么在人工处理的时候发现其完全不可读。

(3) 哈希算法的另一个很广泛的用途就是很多程序员使用它在数据库中保存用户密码。为了保护用户密码的安全性,程序员通常不会直接保存用户密码,而是保存用户密码的哈希值。验证用户密码的时候,用相同的哈希算法计算用户输入的密码,得到哈希值后与数据库中存储的哈希值进行比对,从而完成验证。由于用户密码相同的可能性是很高的,为防止数据库管理员猜测用户密码,程序员还会执行一种俗称“撒盐”的操作,就是在计算用户密码的哈希值之前,把用户密码和另一个比较发散的数据(一般使用用户创建时间的毫秒部分)拼接,这样计算出的哈希值一般都不一样,会很发散。其最大的好处是,即使数据库被泄露,得到用户数据库的黑客也只能看着一大堆哈希值而无可奈何。

2 安全哈希算法的破解

随着科技的发展和密码专家的不断探索、研究,安全哈希算法已经不够安全了。2004 年,对 SHA-0 的解密有突破性的进展,可在 2^{40} 的计算复杂度内找到碰撞。鉴于 SHA-0 的解密成果,密码专家建议那些计划利用 SHA-1 的人们应重新考虑。

2005 年,密码专家发表了对完整版 SHA-1 的攻击结果,只需低于 2^{69} 的计算复杂度,就能找到一组碰撞,而此前利用生日攻击法找到碰撞需要 2^{63} 的计算复杂度。他们展示了一次对 58 次加密循环 SHA-1 的破密,在 2^{33} 的计算复杂度内就找到了一组碰撞。

在密码学的理论中,任何攻击方式,其计算复杂度若低于暴力搜寻法所需要的计算复杂度,就可以被视为针对该密码系统的一种破解法。虽然这并不表示该破解法已经进入实际应用的阶段,但从应用角度来看,一种新的破解法出现,暗示着将来可能会出现更有效率、更接近实用的改良版本。虽然这些实用的破解法版本尚未诞生,但现有的安全哈希算法已经不够安全,确有必要发展更强的哈希算法来取代旧的算法。

4.2.6 XPath 注入漏洞

随着全球化发展,人与人之间的交流日益密切,XML 技术的应用也越来越广泛,例如作为信息传输的载体、应用程序的配置信息或微型数据库等。与此同时,针对 XML 数据信息的 XPath 注入攻击也开始慢慢出现。

可扩展标记语言(eXtensible Markup Language,XML)是一种用于标记电子文件,使其具有结构性的标记语言。XML 文档就是通过 XML 的标准对数据进行描述的一种形

式,它的设计宗旨是传输数据,而不是显示数据。通常情况下,XML 并不具备常见语言的基本功能,即它不能被计算机识别并运行,而需要依靠其他方法(例如 DOM)来帮助计算机提取并识别它。XML 是独立于软件和硬件的信息传输工具,并在近些年成为在各种应用程序之间进行数据传输的最常用的工具。

XPath 即 XML 路径(XML Path)语言,它是一种确定 XML 文档中某部分的位置的语言,可从 XML 文档读取各种信息。XPath 对现有 XML 文档主要是基于 XML 分析器生成节点树的形式来进行分段提取的,生成的树中包含不同类型的节点,包括元素(element)节点、属性(attribute)节点和文本(text)节点等,它提供在数据结构树中寻找节点的能力。XPath 的初衷是作为一个通用的、介于 XPointer(XML Pointer Language,XML 指针语言)与 XSLT(eXtensible Stylesheet Language Transformation,可扩展样式表语言转换)间的语法模型,但是很快 XPath 就被开发者用来当作小型查询语言。

XPath 表达式相当于 XPath 的语言指令,代表其发出的操作指令,其中最重要的一种是路径表达式。XPath 使用路径表达式来选取 XML 文档中的节点或者节点集,在 XML 文档中进行导航,例如,/person/name 就是遍历全部的根元素及其下级的子元素。在 XML 信息被大量使用的今天,其数据的安全性显得非常重要,但是目前关于 XPath 注入攻击防御技术的研究还并不是很多。

1. XPath 注入攻击原理

XPath 注入攻击在一定程度上与 SQL 注入攻击类似,都是输入一些恶意的查询字符串,利用系统没有进行输入检测的漏洞,绕过或修改程序的最初目标功能,从而对网站进行攻击。XPath 注入攻击是指利用 XPath 解析器的输入漏洞和容错特性,在 URL、表单等输入参数中附带恶意的 XPath 查询语句,当这些输入作为参数传入 Web 应用程序时,恶意 XPath 查询语句将被执行,攻击者可以轻易获得敏感信息的访问权并恶意更改这些敏感信息,给系统造成严重的后果。XPath 注入攻击是针对 Web 应用程序的新的攻击方法,它允许攻击者在不知道 XPath 查询相关知识的情况下,通过 XPath 查询得到一个 XML 文档的完整内容。

XPath 注入攻击一般利用两种技术,即 XPath 扫描和 XPath 查询布尔化。通过 XPath 注入攻击,攻击者可以控制用来进行 XPath 查询的 XML 数据库。这种攻击可以有效地利用 XPath 查询(和 XML 数据库)来执行身份验证、查找或者其他操作。XPath 注入攻击同 SQL 注入攻击虽然类似,但和 SQL 注入攻击相比,XPath 注入攻击在以下方面具有优势:

(1) 广泛性。XPath 注入攻击利用的是 XPath 查询,由于 XPath 是一种标准语言,因此,利用 XPath 查询的 Web 应用程序如果未对输入的 XPath 查询进行严格的处理,都会存在 XPath 注入漏洞,所以可能在所有的 XPath 实现中都包含该漏洞。而在 SQL 注入攻击过程中,根据数据库支持的 SQL 语言不同,注入攻击的实现可能不同。

(2) 危害性大。XPath 语言几乎可以引用 XML 文档的所有部分,而对这样的引用一般没有访问控制。在 SQL 注入攻击中,一个用户的权限可能被限制为某一特定的表、列或者查询,而 XPath 注入攻击可以保证得到完整的 XML 文档,即完整的数据库。只要

Web 应用程序存在基本的安全漏洞,即可构造针对 XPath 查询的自动攻击。

2. XPath 注入攻击的防范措施

目前专门的 XPath 注入攻击防御技术还不是太多,一般使用的技术如下:

(1) 当数据提交到服务器端时,在服务器端正式处理这批数据之前,对数据的合法性进行验证。

(2) 检查客户端提交的数据是否包含特殊字符,对特殊字符进行编码转换或替换,删除敏感字符或字符串。

(3) 对于系统出现的错误信息,以 IE 错误编码信息替换,屏蔽系统本身的出错信息。

(4) 参数化 XPath 查询。将需要构建的 XPath 查询表达式以变量的形式表示,变量不是可以执行的脚本。例如,以下代码可以通过创建保存查询的外部文件使查询参数化:

```
declare variable $loginID as xs:string external;
declare variable $password as xs:string external;
//users/user[@loginID=$loginID and @password=$password]
```

(5) 通过 MD5、SSL 等加密算法,对敏感信息和传输过程中的数据进行加密。这样即使非法用户获取数据包,看到的也是加密后的信息。

4.2.7 硬编码密码

在计算机程序或文本编辑中,硬编码是指将变量用一个固定值来代替的方法,也就是把数值写成常数而不是变量。然而,用这种方法编程的程序一旦开始使用,对程序中此变量值的更改将变得十分复杂,程序的可用性将大大降低,因为程序员要找出程序中每一处此变量出现的地方。尽管通过编辑器的查找和替换功能也能实现变量的自动替换,但往往出现多换或者少换的情况,而在计算机程序中,任何小错误的出现都将使程序无法正常运行。

下面用一些简单的程序语句说明硬编码:

- Java 中的硬编码:int a=2,b=2;。
- 硬编码:if(a==2) return false;。
- 非硬编码:if(a==b) return true;。

对密码进行硬编码即在系统中采用明文的形式存储密码。程序员在编写程序时,应尽量避免对密码进行硬编码,而应采用对密码加以模糊化,并且在外部资源文件中进行处理的方法,或将密码先经过哈希处理再存储。硬编码密码会造成任何有该代码权限的人都能读取这个密码,滥用密码现象出现的可能将大大增大。心怀不轨的员工可以利用手中掌握的信息访问权限入侵系统,篡改系统设置。有些人甚至可能利用硬编码密码建立一道后门,从而保证其之后都可以顺利登录并修改部分加密变量。因此,虽然使用硬编码密码可以提高软件开发的效率,但在软件开发完成后将其删去的工作太过于复杂,弊大于利,因此还是应该避免使用硬编码密码。

此外,有一些开发者将密钥硬编码在 Java 代码、文件中,这样做会造成很大风险。信息安全的基础在于密码学,而常用的密码学算法都是公开的,加密内容的保密性依靠的是

密钥的保密性。如果密钥泄露,对于对称密码算法来说,知道了加密用的密钥算法和加密后的密文后,很容易就能得到加密前的明文;对于非对称密码算法或者签名算法来说,通过了解密钥和要加密的明文,可以轻易计算出签名值,从而伪造签名,进行欺诈行为。

4.3 Java 安全编码规范

在 Java 编程语言中,关键的安全编码要素是采用良好的文档和强制的编码规范。本书提供了在 Java 语言中的一系列安全编码规范。这些规范的目标是消除不安全的编码,因为这些不安全的编码会导致可被攻击者利用的漏洞。如果应用这些安全编码规范,可以设计出安全的、可靠的、健壮的、有弹性的、可用性和可维护性高的高质量系统,并且这些安全编码规范还可作为评估源代码质量特性的一个指标(不管是人工还是自动化的过程)。

4.3.1 声明和初始化

1. 防止类的循环初始化

在 Java 语言规范(Java Language Specification,JLS)中将类和接口的初始化描述为:对类进行的初始化包括执行该类的静态初始化方法和初始化该类中的静态数据成员(类变量)。换句话说,一个静态数据成员的出现会触发类的初始化。然而,一个静态数据成员可能会依赖于其他类的初始化,这样有可能形成一个循环初始化。在 JLS 的“类变量的初始化”一节中提到:在运行态中,使用编译期的常量来初始化的用 final 修饰的静态变量是最先初始化的。然而这个描述很容易让人误解,因为对于某些对象实例而言,变量就算是声明为 static final 的,它们的初始化也可能会安排在后期进行,声明一个变量为 static final 并不能够保证它在被读之前已经完全初始化,所以在程序中,特别是在对安全敏感的程序中,必须消除所有的类循环初始化。

1) 类内循环

不符合规范的代码示例如下,它存在涉及多个类的类循环初始化。

```
public class Cycle {
  private final int balance;
  private static final Cycle c = new Cycle();
  private static final int deposit = (int) (Math.random() * 100); // Random deposit

  public Cycle() {
    balance = deposit - 10; // Subtract processing fee
  }

  public static void main(String[] args) {
    System.out.println("The account balance is: " + c.balance);
  }
}
```

在上面的代码中,先定义了一个 final int 变量 balance,静态的初始化方法可以保证只调用一次,而这次调用是在第一次使用静态类变量或者在第一次调用构造函数之前发

生的。接下来，声明了一个与 Cycle 类具有相同属性的 private static final 类变量 c，这个变量会在创建 Cycle 对象的时候初始化。然后进行 deposit 变量的初始化。由于对类变量 c 的初始化在 deposit 变量初始化之前发生，因此，当对变量 c 进行静态初始化时，Cycle 类的构造函数需要读取 deposit 的值。然而，此时由于还没有对 deposit 进行初始化，deposit 的值是 0，而并非程序所需要的随机值，因而最后得到的 balance 的值并不是最初想要的值。

符合规范的代码如下所示：

```
public class Cycle {
  private final int balance;
  private static final int deposit = (int) (Math.random() * 100); // Random deposit
  private static final Cycle c = new Cycle();  // Inserted after initialization of required fields
  public Cycle() {
    balance = deposit - 10; // Subtract processing fee
  }

  public static void main(String[] args) {
    System.out.println("The account balance is: " + c.balance);
  }
}
```

上面的代码改变了类 Cycle 的初始化次序，特别是将变量 c 的初始化插入变量 deposit 的初始化之后，则 c 会在 deposit 被完全初始化之后再进行初始化，所以上面的代码中对数据成员的初始化是不会造成任何依赖循环的。

然而，当涉及许多字段时，开发者很难发现这样的循环初始化。因此，确保不产生这样的循环就非常重要。需要注意的是，尽管这个方案可以防止循环初始化，但它仍然依赖于声明次序，因而也是十分脆弱的。程序的维护者(非开发者)可能不知道这种声明的次序可以保证程序的正确性，所以在代码的文档中需要清楚地说明此处声明次序的重要性。

2) 类间循环

不符合规范的代码示例如下：

```
class A {
  public static final int a = B.b + 1;
  // ...
}

class B {
  public static final int b = A.a + 1;
  // ...
}
```

在程序中声明了两个类，这两个类都有静态变量，而且这些静态变量的值是互相依赖的。当把这两个类放在一起的时候，类间循环问题可以很容易地被发现；然而当把它们分开的时候，开发者却很容易忽略这种错误。

由于对这些类的初始化次序是可变的，所以当初始化次序不同时，程序会计算出不同的 A. a 和 B. b 的值。当先初始化 A 类时，A. a 的值是 2，B. b 的值是 1；而当先初始化 B 类时，A. a 的值是 1，B. b 的值是 2。

符合规范的代码如下：

```
class A {
  public static final int a = 2;
  // ...
}

class B {
  public static final int b = A.a + 1;
  // ...
}
```

通过消除其中一个依赖，即固定一个变量值的初始化值，打破了这个类间循环。程序结果将总是 A. a = 2 且 B. b = 3，与初始化次序无关。

2 不要重用 Java 标准库中公共的标识

不要重用在 Java 标准库中已经使用过的公共的标识，包括公共的工具类、接口或者包。重用公共的标识会降低代码的可读性和可维护性。当开发者使用和公共类相同的名字(如 Vector)定义类后，后来的维护者由于不知道这个标识并不是指 java. util. Vector，可能会无意地使用程序员自定义的 Vector 类而不是原有的 java. util. Vector 类，使得这个由开发者自定义的 Vector 类覆盖了 java. util. Vector 类，从而会导致不可预期的程序行为。

良好定义的 import 语句可以解决这个问题。然而，如果重用的命名定义是从其他包中导入的，使用 type-import-on-demand declaration 会将程序员弄糊涂，因为他需要确定哪一个定义是他想要的。另外，通常程序员会使用 IDE 自动包括 import 语句，一种常见的操作是在编写代码后才生成这些 import 语句，但这种做法容易导致错误。在 Java 包含的 import 引用路径中，如果在预期的类出现之前出现了一个自定义的类，那么将直接选择这个自定义的类，这样程序员就会在无意中使用了错误的类。

不符合规范的代码示例如下：

```
class Vector {
  private int val = 1;

  public boolean isEmpty() {
    if (val == 1) {   // Compares with 1 instead of 0
      return true;
    } else {
      return false;
    }
  }
  // Other functionality is same as java.util.Vector
}

// import java.util.Vector; omitted
public class VectorUser {
  public static void main(String[] args) {
    Vector v = new Vector();
    if (v.isEmpty()) {
      System.out.println("Vector is empty");
    }
  }
}
```

代码中定义了一个类,而这个类重用了 java. util. Vector 的名称。代码在 isEmpty()中使用了一个不同的条件判断,尝试通过重写 java. util. Vector 中对应的方法来达到与遗留代码接口的目的。但如果维护者混淆了这个 isEmpty() 和 java. util. Vector. isEmpty(),就会使程序在运行时出现不可预知的行为。

符合规范的代码如下。

```
class MyVector {
    //other code
}
```

改进后的代码为这个类使用了不同的名字,以防止这个类覆盖 Java 标准类库中的同名类的情况出现。

对于被模仿的原始类,最好的方法就是改变设计策略,这些策略可以更倾向于接口而不是抽象类的方法来实现。将原始类改变成接口,可以让 MyVector 类声明它是假设的接口 Vector 的实现,这可以让使用 MyVector 的代码与使用原始 Vector 实现的代码互相兼容。

3. 将所有增强的 for 语句的循环变量声明为 final 类型

Java 5 平台(因 for-each 循环而出名)引入了增强的 for 语句,用来对对象集合进行迭代。在基本的 for 语句中,给循环变量赋值不会对循环的迭代次序产生影响;在增强的 for 语句中,给循环变量赋值,虽不影响整体的迭代次序,但可能会导致程序员产生困惑,并且会让数据的状态不一致。因此,应避免给 for-each 循环中的循环变量赋值。

一个增强的 for 循环通常采用以下形式:

```
for (Obj Type obj : some Iterable Item) {
    //…
}
```

这等价于以下形式的基本 for 循环:

```
for (Iterator my Iterator = some Iterable Item.iterator();
my Iterator.hasNext();) {
    ObjType obj =myIterator.next();
    //…
}
```

给循环变量赋值等价于修改循环体的局部变量的值,而这个变量的初始值会被循环迭代器引用。这种修改不一定是错误的,但是它可能会使循环功能模糊,也可能表明对增强 for 语句的基础实现有误解。可以将 for 语句中的所有循环变量声明为 final,从而拒绝对这个循环变量的任何赋值。

不符合规范的代码示例如下:

```
List<Integer> list = Arrays.asList(new Integer[] {13, 14, 15});
boolean first = true;

System.out.println("Processing list...");
for (Integer i: list) {
  if (first) {
    first = false;
    i = new Integer(99);
  }

  System.out.println(" New item: " + i);
  // Process i
}

System.out.println("Modified list?");
for (Integer i: list) {
  System.out.println("List item: " + i);
}
```

上面的代码示例使用增强的 for 循环处理对象集合。此外，它还希望跳过对集合中某一个元素的处理。

这种跳过下一个集合元素的想法看起来是可以实现的，因为已经成功地为循环变量赋值，并且 processMe 变量的值也更新了。然而，与基本的 for 循环不同，这个赋值并没有改变循环执行的迭代次序。因此，虽然需要跳过的元素跳过了，但是它之后的元素被处理了两次。注意，如果声明 processMe 为 final 型，在试图对它进行这样的赋值时，就会产生编译器错误。

符合规范的代码如下：

```
// ...
for (final Integer i: list) {

// ...

for (final Integer i: list) {
  Integer item = i;
  if (first) {
    first = false;
    item = new Integer(99);
  }
  System.out.println(" New item: " + item);
  // Process item
}
public void deleteFile(){

  File someFile = new File("someFileName.txt");
  // Do something with someFile
  if (!someFile.delete()) {
    // Handle failure to delete the file
  }

}
```

这个符合规范的代码可以正确地处理集合中的每一个对象，并且每一个对象只会被处理一次。

4.3.2　表达式

1. 不要忽略方法的返回值

如果程序需要通过返回值来判断方法调用成功与否，或者需要通过返回值来更新对象或数据成员，当忽略方法的返回值时，可能导致不可预期的程序行为。

不符合规范的代码示例如下：

```
public void deleteFile(){

  File someFile = new File("someFileName.txt");
  // Do something with someFile
  someFile.delete();

}
```

符合规范的代码示例如下：

```
public void deleteFile(){

  File someFile = new File("someFileName.txt");
  // Do something with someFile
  if (!someFile.delete()) {
    // Handle failure to delete the file
  }

}
```

不符合规范的代码示例如下：

```
public class Replace {
  public static void main(String[] args) {
    String original = "insecure";
    original.replace('i', '9');
    System.out.println(original);
  }
}
```

符合规范的代码示例如下：

```
public class Replace {
  public static void main(String[] args) {
    String original = "insecure";
    original = original.replace('i', '9');
    System.out.println(original);
  }
}
```

2. 不要解引用空指针

在需要对象的实例中，不要使用 null 值，包括：调用 null 对象的实例方法，访问或修

改 null 对象的字段，取数组为 null 的长度，访问或修改数组元素为 null 的对象，抛出的异常对象为 null，等等。

不符合规范的代码示例如下：

```
public static int cardinality(Object obj, final Collection<?> col) {
  int count = 0;
  if (col == null) {
    return count;
  }
  Iterator<?> it = col.iterator();
  while (it.hasNext()) {
    Object elt = it.next();
    if ((null == obj && null == elt) || obj.equals(elt)) {  // Null pointer dereference
      count++;
    }
  }
  return count;
}
```

符合规范的代码示例如下：

```
public static int cardinality(Object obj, final Collection col) {
  int count = 0;
  if (col == null) {
    return count;
  }
  Iterator it = col.iterator();
  while (it.hasNext()) {
    Object elt = it.next();
    if ((null == obj && null == elt) ||
        (null != obj && obj.equals(elt))) {
      count++;
    }
  }
  return count;
}
```

3. 不要使用 equals()比较两个数组的内容

使用 equals()比较两个数组常被误解为比较的是两个数组的内容，实际上 equals()方法比较的是数组的引用而不是数组的内容。要比较两个数组的内容，需要使用 Arrays. equals()。而如果使用 equals()方法比较数组的内容，会产生不正确的结果，从而导致安全问题。

不符合规范的代码示例如下：

```
int[] arr1 = new int[20]; // Initialized to 0
int[] arr2 = new int[20]; // Initialized to 0
System.out.println(arr1.equals(arr2)); // Prints false
```

符合规范的代码示例如下：

```
int[] arr1 = new int[20]; // Initialized to 0
int[] arr2 = new int[20]; // Initialized to 0
System.out.println(Arrays.equals(arr1, arr2)); // Prints true
```

符合规范的代码示例如下：

```
int[] arr1 = new int[20]; // Initialized to 0
int[] arr2 = new int[20]; // Initialized to 0
System.out.println(arr1 == arr2); // Prints false
```

4. 不要用== 和!=比较基本数据类型的封装值

不要用＝＝和！＝对基本数据类型的封装值进行比较，因为这两个操作符比较的是对象引用而不是对象值。如果自动封装的值是 true 和 false、从\u0000 到\u007f 的字符、－128～127 的整型或短整型值，那么用＝＝和！＝进行比较，结果是成立的。使用＝＝和！＝操作符比较基本数据类型的封装值，会导致错误的比较结果。

不符合规范的代码示例如下：

```
public class Wrapper {
  public static void main(String[] args) {
    Integer i1 = 100;
    Integer i2 = 100;
    Integer i3 = 1000;
    Integer i4 = 1000;
    System.out.println(i1 == i2);
    System.out.println(i1 != i2);
    System.out.println(i3 == i4);
    System.out.println(i3 != i4);
  }
}
```

符合规范的代码示例如下：

```
public class Wrapper {
  public static void main(String[] args) {
    Integer i1 = 100;
    Integer i2 = 100;
    Integer i3 = 1000;
    Integer i4 = 1000;
    System.out.println(i1.equals(i2));
    System.out.println(!i1.equals(i2));
    System.out.println(i3.equals(i4));
    System.out.println(!i3.equals(i4));
  }
}
```

不符合规范的代码示例如下：

```
public void exampleEqualOperator(){
  Boolean b1 = new Boolean("true");
  Boolean b2 = new Boolean("true");

  if (b1 == b2) {     // Never equal
    System.out.println("Never printed");
  }
}
```

符合规范的代码示例如下：

```
public void exampleEqualOperator(){
  Boolean b1 = true;
  Boolean b2 = true;

  if (b1 == b2) {   // Always equal
    System.out.println("Always printed");
  }

  b1 = Boolean.TRUE;
  if (b1 == b2) {   // Always equal
    System.out.println("Always printed");
  }
}
```

5. 不要将不同类型的参数传给特定 Java 集合框架方法

Java 集合框架方法有采用泛型定义的，也有不采用泛型定义的，这种设计是为了更好地兼容，但是这种设计也会导致编码错误。将参数传递给特定的 Java 集合框架方法，当参数类型与方法类型不一致时，可能会导致失败，从而导致意外的对象保留、内存泄漏或不正确的程序操作等。

不符合规范的代码示例如下：

```
public class ShortSet {
  public static void main(String[] args) {
    HashSet<Short> s = new HashSet<Short>();
    for (int i = 0; i < 10; i++) {
      s.add((short)i); // Cast required so that the code compiles
      s.remove(i); // Tries to remove an Integer
    }
    System.out.println(s.size());
  }
}
```

符合规范的代码示例如下：

```
public class ShortSet {
  public static void main(String[] args) {
    HashSet<Short> s = new HashSet<Short>();
    for (int i = 0; i < 10; i++) {
      s.add((short)i);
      // Remove a Short
      if (s.remove((short)i) == false) {
        System.err.println("Error removing " + i);
      }
    }
    System.out.println(s.size());
  }
}
```

6. 不要在断言中使用有副作用的表达式

断言是一种方便的机制，可以用来在代码中加入诊断测试。使用标准断言语句的表达式时必须避免副作用，此副作用会导致程序行为依赖于断言功能是否已开启。

不符合规范的代码示例如下：

```
private ArrayList<String> names;

void process(int index) {
  assert names.remove(null); // Side effect
  // ...
}
```

符合规范的代码示例如下：

```
private ArrayList<String> names;

void process(int index) {
  boolean nullsRemoved = names.remove(null);
  assert nullsRemoved; // No side effect
  // ...
}
```

4.3.3　面向对象

1. 限制字段的访问

将数据成员声明为私有，则会破坏程序封装性，同时将该数据成员暴露给其他程序。

不符合规范的代码示例如下：

```
public class Widget {

    public int total; // Number of elements
    void add() {
        if (total < Integer.MAX_VALUE) {
            total++;
            // ...
        } else {
            throw new ArithmeticException("Overflow");
        }
    }
    void remove() {
        if (total > 0) {
            total--;
            // ...
        } else {
            throw new ArithmeticException("Overflow");
        }
    }
}
```

符合规范的代码示例如下：

```
public class Widget {

    private int total; // Declared private
    public int getTotal() {
        return total;
    }
    void add() {
        if (total < Integer.MAX_VALUE) {
            total++;
            // ...
        } else {
            throw new ArithmeticException("Overflow");
        }
    }
    void remove() {
        if (total > 0) {
            total--;
            // ...
        } else {
            throw new ArithmeticException("Overflow");
        }
    }
}
```

2 防止堆污染

当参数化类型的变量引用不属于参数化类型的对象时，就会发生堆污染。将泛型类型的代码与原始类型的代码混合在一起是堆污染的一个常见来源。泛型类型是在 Java 5 引入的，对于遗留代码的改进会涉及这个问题。

不符合规范的代码示例如下，它将错误的数据加入列表中。

```
public class ListUtility {

    private static void addToList(List list, Object obj) {
        list.add(obj); // Unchecked warning
    }
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        addToList(list, 42);
        System.out.println(list.get(0)); // Throws ClassCastException
    }

}
```

符合规范的代码示例如下：

```
 private static void addToList(List<String> list, String str) {
     list.add(str); // No warning generated
 }

 public static void main(String[] args) {
     List<String> list = new ArrayList<String>();
     addToList(list, "42");
```

```
        System.out.println(list.get(0));
    }

    private static void addToList(List list, Object obj) {
        list.add(obj); // Unchecked warning, also throws ClassCastException
    }
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        List<String> checkedList = Collections.checkedList(list, String.class);
        addToList(checkedList, 42);
        System.out.println(list.get(0));
    }
```

3. 不要使用公共静态的非 final 类型字段

在程序中不要定义非 final 类型的公共静态(public static)字段,这样的字段可被外部类更改。

不符合规范的代码示例如下:

```
package org.apache.xpath.compiler;

public class FunctionTable {
  public static FuncLoader m_functions;
}
```

符合规范的代码示例如下:

```
public static final FuncLoader m_functions;
// Initialize m_functions in a static initialization block
```

4.3.4　方法

1. 必要的方法参数验证

应该对方法的参数进行验证,从而确保参数是在方法预计的范围内,保证基于该参数的操作能产生有效的结果。如果没有对参数进行验证,可能会导致不一致的运算、执行异常和控制流漏洞。

不符合规范的代码示例如下:

```
private Object myState = null;

// Sets some internal state in the library
void setState(Object state) {
  myState = state;
}

// Performs some action using the state passed earlier
void useState() {
  // Perform some action here
}
```

符合规范的代码示例如下：

```
private Object myState = null;

// Sets some internal state in the library
void setState(Object state) {
  if (state == null) {
    // Handle null state
  }

  // Defensive copy here when state is mutable

  if (isInvalidState(state)) {
    // Handle invalid state
  }
  myState = state;
}

// Performs some action using the state passed earlier
void useState() {
  if (myState == null) {
    // Handle no state (e.g., null) condition
  }
  // ...
}
```

2. 不要使用弃用的或过时的类和方法

不要在新开发的代码中使用弃用的或过时的类和方法，否则会导致不正确的程序行为。弃用的或过时的类和方法的替换示例如表 4-1 所示。

表 4-1　弃用的或过时的类和方法的替换示例

弃用的或过时的类和方法	新的类和方法
java. lang. Character. isJavaLetter()	java. lang. Character. isJavaIdentifierStart()
java. lang. Character. isJavaLetterOrDigit()	java. lang. Character. isJavaIdentifierPart()
java. lang. Character. isSpace()	java. lang. Character. isWhitespace()
java. lang. Class. newInstance()	java. lang. reflect. Constructor. newInstance()
java. util. Date(many methods)	java. util. Calendar
java. util. Dictionary<K,V>	java. util. Map<K,V>

3. 不要在 clone()方法中调用可重写的方法

在 clone()方法中调用可重写的方法是不安全的，一个恶意的子类可以对方法进行重写并改变 clone()的行为，一个可信任的子类可以查看被克隆对象在创建完成之前的某一特定的初始化状态。在克隆过程中调用可重写的方法会将类的内部暴露给恶意代码，或将特定的初始化状态暴露给可信任代码，从而违反类的不变性。

4.3.5　异常处理

1. 不要消除或忽略可检查异常

程序员常常使用一个空的或简单的 catch 程序段来捕获受控制的异常，并将其消除。每一个 catch 程序段都必须确保程序只在功能不变且有效的情况下继续运行。因此，catch 程序段必须要么从异常中恢复，将异常重新抛出，由下一个和 try 语句对应的 catch 程序段来处理异常；要么根据 catch 程序段的上下文抛出另一个适合的异常。这种行为的风险是消除或忽略异常会导致不一致的程序状态。

AMQ-1727 描述了 ActiveMQ 服务中的一个漏洞，当 ActiveMQ 从 STOMP 客户端接收了无效的用户名和密码时生成了一个安全异常，但是被忽略了，这使得 STOMP 客户端可以对 ActiveMQ 进行完全不受限制的访问。

不符合规范的代码示例如下：

```
try {
  //...
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

打印异常的堆栈跟踪可以用于调试目的，但是执行生成的程序等同于抑制异常。打印堆栈跟踪还可以向攻击者泄露有关该进程的结构和状态的信息。注意，即使这个不兼容的代码示例通过打印堆栈跟踪来对异常作出反应，但它仍然会继续执行，就好像这个异常没有被抛出一样。也就是说，应用程序的行为不受抛出异常的影响，除非在抛出异常的点之后，在 try 块中出现的任何表达式或语句都不会被评估。

下面的代码通过请求用户指定另一个文件名来处理一个 FileNotFoundException 异常：

```
volatile boolean validFlag = false;
do {
  try {
    // If requested file does not exist, throws FileNotFoundException
    // If requested file exists, sets validFlag to true
    validFlag = true;
  } catch (FileNotFoundException e) {
    // Ask the user for a different file name
  }
} while (validFlag != true);
// Use the file
```

2. 不能允许异常泄露敏感信息

异常可能会泄露敏感信息。在异常传递的过程中，如果不对敏感信息进行过滤，常会导致信息泄露，这将有助于攻击者实现对系统的攻击。

CVE-2009-2897 描述了在 SpringSource Hyperic HQ 几个版本中的跨站脚本漏洞，这些漏洞允许远程攻击者通过赋予参数无效的数据来注入任意网页脚本或 HTML 代码。这些

漏洞源于一个未捕获的 java. lang. NumberFormatException 异常，而这个异常是由在网站界面中输入无效数值引起的。异常导致的信息泄露或者威胁的示例如表 4-2 所示。

表 4-2　异常导致的信息泄露或者威胁的示例

示 例 名 称	信息泄漏或威胁的描述
java. io. FileNotFoundException	底层文件系统结构，用户名枚举
java. lang. OutOfMemoryError	拒绝服务
java. lang. StackOverflowError	拒绝服务
java. net. BindException	不受信任的客户端可以在选择服务器端口时打开端口的枚举
java. security. acl. NotOwnerException	所有者枚举
java. sql. SQLException	数据库结构，用户名枚举
java. util. ConcurrentModificationException	可能提供有关线程不安全代码的信息
java. util. jar. JarException	底层文件系统结构
java. util. MissingResourceException	资源枚举
javax. naming. InsufficientResourcesException	服务器资源不足（可能有助于 DoS）

不符合规范的代码示例如下：

```
class ExceptionExample {
  public static void main(String[] args) throws FileNotFoundException {
    // Linux stores a user's home directory path in
    // the environment variable $HOME, Windows in %APPDATA%
    FileInputStream fis =
        new FileInputStream(System.getenv("APPDATA") + args[0]);
  }
}
```

在上面的代码示例中，程序必须读取用户提供的文件，但是文件系统的内容和结构是敏感的。程序接收一个文件名作为输入参数，但是不能阻止任何产生的异常被提交给用户。当一个请求的文件没有被找到时，FileInputStream 构造函数抛出一个 FileNotFoundException，允许攻击者通过重复向程序传递虚假的路径名来重建底层文件系统。

符合规范的代码示例如下：

```
try {
  FileInputStream fis = new FileInputStream(file);
} catch (FileNotFoundException x) {
  System.out.println("Invalid file");
  return;
}
```

3. 记录日志时应避免异常

记录日志时抛出的异常会妨碍记录的完成。如果没有考虑到写入日志时发生异常的

可能性,则会造成安全漏洞。

将此类异常写入标准错误流,对于日志记录来说是不够的。首先,标准错误流可能会被耗尽或关闭,从而防止后续异常的记录。其次,标准错误流的信任级别可能不足以记录某些安全关键的(security-critical)异常或错误,而不会泄露敏感信息。如果在编写安全异常时出现了一个输入输出错误,那么 catch 块将抛出一个 IOException,而临界安全异常将丢失。最后,攻击者可能会伪造异常,使其发生在其他几个无害的异常中。使用 Console. printf()、System. out. print()或 Throwable. printStackTrace()输出一个安全异常也违反了这条规范。在对数据进行日志记录时,抛出的异常会导致数据丢失。

HARMONY-59812 描述了以下漏洞:FileHandler 类接收到日志信息,但如果一个线程关闭了对应的文件,则另一个线程会在尝试记录日志时抛出异常。

不符合规范的代码示例如下:

```
try {
  // ...
} catch (SecurityException se) {
  System.err.println(se);
  // Recover from exception
}
```

符合规范的代码示例如下:

```
try {
  // ...
} catch(SecurityException se) {
  logger.log(Level.SEVERE, se);
  // Recover from exception
}
```

4. 不要在 finally 程序段非正常退出

不要在 finally 程序段中使用 return、break、continue 或 throw 语句。当程序进入带有 finally 程序段的 try 程序段时,不管 try 程序段是否完成执行,finally 程序段总是会执行的,造成 finally 程序段非正常终止的语句会导致 try 程序段非正常终止,从而消除了从 try 或 catch 程序段抛出的任何异常。非正常退出 finally 程序段掩盖了对应的 try 或 catch 程序段抛出的任何异常。

不符合规范的代码示例如下:

```
class TryFinally {
  private static boolean doLogic() {
    try {
      throw new IllegalStateException();
    } finally {
      System.out.println("logic done");
      return true;
    }
  }
}
```

符合规范的代码示例如下：

```
class TryFinally {
  private static boolean doLogic() {
    try {
      throw new IllegalStateException();
    } finally {
      System.out.println("logic done");
    }
    // Any return statements must go here;
    // applicable only when exception is thrown conditionally
  }
}
```

5. 不要捕获 NullPointerException

程序不能捕获 java.lang.NullPointerException，一个运行时抛出的空指针异常表明存在一个对空指针的解引用，这是必须在应用程序中解决的。捕获空指针异常可能掩盖潜在的空指针解引用，降低应用性能，并造成代码难以理解和维护。RuntimeException、Exception 和 Throwable 等也可能会掩盖其他异常，并妨碍对异常的正确处理。

不符合规范的代码示例如下：

```
boolean isName(String s) {
  try {
    String names[] = s.split(" ");

    if (names.length != 2) {
      return false;
    }
    return (isCapitalized(names[0]) && isCapitalized(names[1]));
  } catch (NullPointerException e) {
    return false;
  }
}
```

符合规范的代码示例如下：

```
boolean isName(String s) {
  if (s == null) {
    return false;
  }
  String names[] = s.split(" ");
  if (names.length != 2) {
    return false;
  }
  return (isCapitalized(names[0]) && isCapitalized(names[1]));
}
```

4.3.6　线程锁

1. 不要将锁对象暴露给与非受信代码交互的类

声明为 synchronized 的方法和同步于 this 引用的代码块都使用了对象的 monitor（内置锁），攻击者会通过控制系统来触发竞争和死锁，这可以通过获得并且永久地持有一个可访问的类的内置锁来实现，从而引发 DoS 攻击。

不符合规范的代码示例如下：

```
public class SomeObject {
  //changeValue locks on the class object's monitor
  public static synchronized void changeValue() {
    // ...
  }
}

// Untrusted code
synchronized (SomeObject.class) {
  while (true) {
    Thread.sleep(Integer.MAX_VALUE); // Indefinitely delay someObject
  }
}
```

符合规范的代码示例如下：

```
public class SomeObject {
  private static final Object lock = new Object();

  public static void changeValue() {
    synchronized (lock) { // Locks on the private Object
      // ...
    }
  }
}
```

2. 不要基于可能被重用的对象进行同步

错误地使用同步的基本数据类型是并发问题的一个常见来源，基于可能被重用的对象进行同步，会导致死锁和不确定的程序行为。使用错误的对象类型进行同步是产生并发漏洞的主要原因。

不符合规范的代码示例如下：

```
private final String lock = new String("LOCK").intern();

public void doSomething() {
  synchronized (lock) {
    // ...
  }
}
```

符合规范的代码示例如下：

```
private final Object lock = new Object();

public void doSomething() {
  synchronized (lock) {
    // ...
  }
}
```

3. 不要基于 getClass()返回的类对象来实现同步

基于 getClass()的返回值进行同步，会导致不可预期的程序行为。对于子类的 Class 对象来说，它与其基类的 Class 对象是完全不同的。

不符合规范的代码示例如下：

```
class Base {
  static DateFormat format =
      DateFormat.getDateInstance(DateFormat.MEDIUM);

  public Date parse(String str) throws ParseException {
    synchronized (getClass()) { // Intend to synchronizes on Base.class
      return format.parse(str);
    }
  }

  public Date doSomething(String str) throws ParseException {
    return new Helper().doSomethingAndParse(str);
  }

  private class Helper {
    public Date doSomethingAndParse(String str) throws ParseException {
      synchronized (getClass()) { // Synchronizes on Helper.class
        // ...
        return format.parse(str);
      }
    }
  }
}
```

符合规范的代码示例如下：

```
class Base {
  static DateFormat format =
      DateFormat.getDateInstance(DateFormat.MEDIUM);

  public Date parse(String str) throws ParseException {
```

```
    try {
      synchronized (Class.forName("Base")) {
        return format.parse(str);
      }
    } catch (ClassNotFoundException x) {
      // "Base" not found; handle error
    }
    return null;
  }
}
```

4. 不要基于 high-level 并发对象的内置锁来实现同步

在 java. util. concurrent. locks 包中的 Lock 和 Condition 接口的类，被认为是 high-level 的并发对象，对这样的对象使用内置锁是有问题的做法，尽管看起来代码的功能是正确的。这样做容易造成混乱：使用锁对象的代码很可能与使用锁接口的代码交互，这导致两个组件误认为它们用来保护数据的锁是用相同的，而实际上它们使用两个不同的锁。因此，锁将无法保护任何数据。基于 high-level 并发类库的内置锁实现同步，会因为不一致的锁规则而导致不确定的行为。

不符合规范的代码示例如下：

```
private final Lock lock = new ReentrantLock();

public void doSomething() {
  synchronized(lock) {
    // ...
  }
}
```

符合规范的代码示例如下：

```
private final Lock lock = new ReentrantLock();

public void doSomething() {
  lock.lock();
  try {
    // ...
  } finally {
    lock.unlock();
  }
}
```

4.3.7　线程 API

1. 不要直接调用 Thread.run()

规范描述：启动一个新线程应该调用 Thread 的 start()方法。如果直接调用 Thread 的 run()方法，run()方法中的语句是由当前线程而不是新创建的线程来执行的。不能正

确地启动线程会导致不可预期的程序行为。

不符合规范的代码示例如下：

```
public final class Foo implements Runnable {
  @Override public void run() {
    // ...
  }

  public static void main(String[] args) {
    Foo foo = new Foo();
    new Thread(foo).run();
  }
}
```

符合规范的代码示例如下：

```
public final class Foo implements Runnable {
  @Override public void run() {
    // ...
  }

  public static void main(String[] args) {
    Foo foo = new Foo();
    new Thread(foo).start();
  }
}
```

2. 不要调用 ThreadGroup 中的一些方法

Java 中的每个线程都属于一个线程组，ThreadGroup 提供了对于同一线程组的线程同时进行操作的便利方法。但 ThreadGroup 中的很多方法是不提倡使用的（如 resume()、stop()、suspend()等），还有一些危险但可以使用的方法，如 ThreadGroup. activeCount()、ThreadGroup. enumerate()。使用 ThreadGroup API 可能导致竞态、内存泄漏及不一致的对象状态。

3. 唤醒所有等待中的线程而不是一个线程

Object 的 notify()和 notifyAll()分别用来唤醒等待中（调用了 Object. wait()方法）的一个或多个线程。notifyAll()方法会唤醒所有等待同一对象锁的线程，notify()只唤醒一个线程，并不保证哪一个线程会接到通知。如果没有满足等待的条件，那么选中的线程会继续等待，这样就违背了通知的目的。同样 java. util. concurrent. locks 的 Condition. signal()和 Condition. signalAll()用来唤醒调用时由于 Condition. await()受阻的线程。

4. 不要使用 Thread.stop()来终止线程

调用该方法将导致释放该线程持有的所有锁，可能会暴露这些锁保护的对象。

不符合规范的代码示例如下：

```
public final class Container implements Runnable {
  private final Vector<Integer> vector = new Vector<Integer>(1000);

  public Vector<Integer> getVector() {
    return vector;
  }

  @Override public synchronized void run() {
    Random number = new Random(123L);
    int i = vector.capacity();
    while (i > 0) {
      vector.add(number.nextInt(100));
      i--;
    }
  }

  public static void main(String[] args) throws InterruptedException {
    Thread thread = new Thread(new Container());
    thread.start();
    Thread.sleep(5000);
    thread.stop();
  }
}
```

在展示符合规范的例子之前,先对中断线程 interrupt()进行描述:

当对一个线程调用 interrupt()时,不是真正中断了正在运行的线程,而是将线程的中断状态(每个线程都有的布尔标志位)置位,每个线程都应该不时地检查这个标志位,以判断线程是否被中断。

如果一个线程处于阻塞状态(如 thread. sleep、thread. join、thread. wait 等),则在检查中断标志位时如果发现中断标志位为 true,则会在这些阻塞方法调用处抛出 InterruptedException 异常,并且在抛出异常后立即将线程的中断标志位清除,即重新设置为 false。抛出异常是为了将线程从阻塞状态唤醒,并在结束线程前让程序员有足够的时间来处理中断请求。

不可中断的操作,包括进入 synchronized 段以及 Lock. lock()、inputStream. read()等,调用 interrupt()无效,因为它们都不抛出中断异常。如果不能获得资源,它们会无限期阻塞。对于 Lock. lock(),可以改用 Lock. lockInterruptibly(),即可被中断的加锁操作,它可以抛出中断异常。对于 inputStream 等资源,可以通过 close()方法将其关闭,对应的阻塞也会被解除。

符合规范的代码示例如下:

```
public final class Container implements Runnable {
  private final Vector<Integer> vector = new Vector<Integer>(1000);
  private volatile boolean done = false;

  public Vector<Integer> getVector() {
    return vector;
  }
```

```
  public void shutdown() {
    done = true;
  }

  @Override public synchronized void run() {
    Random number = new Random(123L);
    int i = vector.capacity();
    while (!done && i > 0) {
      vector.add(number.nextInt(100));
      i--;
    }
  }

  public static void main(String[] args) throws InterruptedException {
    Container container = new Container();
    Thread thread = new Thread(container);
    thread.start();
    Thread.sleep(5000);
    container.shutdown();
  }
}
```

线程的 thread.interrupt()方法用于中断线程，将设置该线程的中断标志位，即设置为 true，中断的结果是线程死亡还是线程等待新的任务或继续运行至下一步，取决于程序本身。线程会不时地检测中断标志位，以判断线程是否应该被中断(中断标志位是否为 true)，它并不像 stop()方法那样会中断一个正在运行的线程。

下面是使用 volatile 标志位来请求线程终止的代码：

```
public final class Container implements Runnable {
  private final Vector<Integer> vector = new Vector<Integer>(1000);

  public Vector<Integer> getVector() {
    return vector;
  }

  @Override public synchronized void run() {
    Random number = new Random(123L);
    int i = vector.capacity();
    while (!Thread.interrupted() && i > 0) {
      vector.add(number.nextInt(100));
      i--;
    }
  }

  public static void main(String[] args) throws InterruptedException {
    Container c = new Container();
    Thread thread = new Thread(c);
    thread.start();
    Thread.sleep(5000);
    thread.interrupt();
  }
}
```

4.3.8　输入输出

1. 检查和处理与文件相关的错误

Java 的文件操作方法通常使用返回值而不是抛出异常来指示其错误，因此，程序如果忽略文件操作返回值，将不能发现文件操作的失败。不对执行文件 I/O 操作的返回值进行检查，可能会产生异常的行为。

因为 SecurityException 是一个运行时异常，所以不需要声明它。NoSuchFileException、DirectoryNotExmptyException 和 IOException 都属于编译时异常，需要处理。NoSuchFileException 和 DirectoryNotExmptyException 在继承 IOException 的同时也继承了 IOException 中 catch 子句的解决方案。

不符合规范的代码示例如下：

```
public void methodBad01(String args[]) {
    File file = new File(args[0]);
    file.delete();
}
```

符合规范的代码示例如下：

```
public void methodGood01() {
    File file = new File("file");
    if (!file.delete()) {
        System.out.println("Deletion failed");
    }
}
```

2. 在程序终止前删除临时文件

在不再需要临时文件时应删除它们，可以让文件名和其他资源循环使用。每个程序都应在正常操作期间删除临时文件。没有绝对的方法可以保证在异常终止的情况下删除孤立的文件，即使使用 finally 代码块也不能保证这一点。出于这个原因，许多系统使用临时文件清理工具来扫描临时目录和删除旧文件。这样的实用程序可以由系统管理员手动调用，也可以由系统进程定期调用。但是，这些工具本身也常常存在基于文件操作的安全漏洞。在程序终止之前未删除临时文件会导致信息泄露和资源耗尽。

不符合规范的代码示例如下：

```
public static void main(String[] args) throws IOException {
    File f = new File("tempnam.tmp");
    if (f.exists()) {
        System.out.println("This file already exists");
        return;
    }
    FileOutputStream fop = null;
    try {
        fop = new FileOutputStream(f);
        String str = "Data";
        fop.write(str.getBytes());
    } finally {
        if (fop != null) {
            try {
                fop.close();
```

```
                } catch (IOException x) {
                    // Handle error
                }
            }
        }
}
```

符合规范的代码示例如下：

```
public static void main(String[] args) {
    Path tempFile = null;
    try {
        tempFile = Files.createTempFile("tempnam", ".tmp");
        try (BufferedWriter writer =
                Files.newBufferedWriter(tempFile, Charset.forName("UTF8"),
                    StandardOpenOption.DELETE_ON_CLOSE)) {
            // Write to file
        }
        System.out.println("Temporary file write done, file erased");
    } catch (FileAlreadyExistsException x) {
        System.err.println("File exists: " + tempFile);
    } catch (IOException x) {
        // Some other sort of failure, such as permissions.
        System.err.println("Error creating temporary file: " + x);
```

代码中使用了 Java 7 的 nio2 包中的方法创建临时文件，createTempFile()方法创建了一个不可预测的名字，且代码中使用 try 的构造函数来打开文件，这样不管是否出现异常，系统都会自动关闭文件。最后使用 DELETE_ON_CLOSE 选项，以实现在关闭文件的时候删除文件。

3. 检查和处理与文件相关的错误

调用 Java 垃圾收集器释放没有被引用且没有被释放的内存。但是，Java 垃圾收集器不能释放非内存资源，例如打开文件描述符和数据库连接。当资源不再需要时，如果不能直接释放这些非内存资源，会耗尽这些资源。

不符合规范的代码示例如下：

```
public int processFile(String fileName)
                        throws IOException, FileNotFoundException {
  FileInputStream stream = new FileInputStream(fileName);
  BufferedReader bufRead =
      new BufferedReader(new InputStreamReader(stream));
  String line;
  while ((line = bufRead.readLine()) != null) {
    sendLine(line);
  }
  return 1;
}
```

符合规范的代码示例如下：

```
try {
  final FileInputStream stream = new FileInputStream(fileName);
  try {
    final BufferedReader bufRead =
        new BufferedReader(new InputStreamReader(stream));
```

```
      String line;
      while ((line = bufRead.readLine()) != null) {
        sendLine(line);
      }
    } finally {
      if (stream != null) {
        try {
          stream.close();
        } catch (IOException e) {
          // Forward to handler
        }
      }
    }
  } catch (IOException e) {
    // Forward to handler
  }
```

```
try (FileInputStream stream = new FileInputStream(fileName);
     BufferedReader bufRead =
         new BufferedReader(new InputStreamReader(stream))) {

    String line;
    while ((line = bufRead.readLine()) != null) {
      sendLine(line);
    }
  } catch (IOException e) {
    // Forward to handler
  }
```

4. 使用 wrap()或 duplicate()方法创建的缓冲区不要暴露给不受信任的代码

Buffer 类在 java. nio 包中定义，如 IntBuffer、CharBuffer、ByteBuffer，这些类定义了一系列的 wrap()方法，用来将对应的基本数据类型的数据封装到一个缓冲区中，并返回一个 Buffer 对象来代表这个缓冲区。新的缓冲区由指定的字符数组来支持，对这些缓冲区的修改会导致数组的修改。将这些缓冲区暴露给不受信任的代码，会将其背后的数组也暴露给恶意的攻击者。同样，duplicate()方法创建了额外的缓冲区，这个缓冲区背后是支持原始缓冲区的数组。暴露这个额外的缓冲区给不受信任的代码，同样也会将其暴露给恶意的攻击者。使用 wrap()或 duplicate()方法创建的缓冲区可能会暴露给不受信任的调用者，使其可以修改原始数据。

不符合规范的代码示例如下：

```
final class Wrap {
  private char[] dataArray;

  public Wrap() {
    dataArray = new char[10];
    // Initialize
  }
```

```
    public CharBuffer getBufferCopy() {
      return CharBuffer.wrap(dataArray);
    }
}
```

上面的代码示例声明了一个 char 数组，将其封装在缓冲区 CharBuffer 中，并通过 getBufferCopy()方法将 CharBuffer 暴露给不受信任的代码。

符合规范的代码示例如下：

```
final class Wrap {
  private char[] dataArray;

  public Wrap() {
    dataArray = new char[10];
    // Initialize
  }

  public CharBuffer getBufferCopy() {
    return CharBuffer.wrap(dataArray).asReadOnlyBuffer();
  }
}
```

不符合规范的代码示例如下：

```
final class Dup {
  CharBuffer cb;

  public Dup() {
    cb = CharBuffer.allocate(10);
    // Initialize
  }

  public CharBuffer getBufferCopy() {
    return cb.duplicate();
  }
}
```

上面的代码示例调用 duplicate()方法来创建和返回 CharBuffer 的副本，返回的缓冲区由与原始缓冲区相同的数组支持。因此，如果调用者修改了支持数组的元素，那么这些修改也会影响原始缓冲区。

符合规范的代码示例如下：

```
final class Dup {
  CharBuffer cb;

  public Dup() {
    cb = CharBuffer.allocate(10);
    // Initialize
  }

  public CharBuffer getBufferCopy() {
    return cb.asReadOnlyBuffer();
  }
}
```

5. 从一个流中读取字符或字节与−1 的比较

Inputstream.read()和 Reader.read()方法分别用于从一个流中读取一个字节或字符。Inputstream.read()从一个输入源读取一字节，并将其值作为一个 0～255 的整型值返回。Reader.read()方法读取一个字符，并将其值作为一个 0～65 535 的整型值返回。这两个方法都通过返回 32 位值−1 表示已经到达流的结束处，后面再没有可用的数据。如果在与值−1 进行比较之前将结果由整型值转换为字节或字符，将会产生错误。程序必须在流结束后再将返回值压缩到字节或字符中。使用一个窄类型的类获取字节输入方法的返回值会导致严重的漏洞，包括命令行注入漏洞。

不符合规范的代码示例如下：

```
FileInputStream in;
// Initialize stream
byte data;
while ((data = (byte) in.read()) != -1) {
  // ...
}
```

符合规范的代码示例如下：

```
FileReader in;
// Initialize stream
int inbuff;
char data;
while ((inbuff = in.read()) != -1) {
  data = (char) inbuff;
  // ...
}
```

不符合规范的代码示例如下：

```
FileReader in;
// Initialize stream
char data;
while ((data = (char) in.read()) != -1) {
  // ...
}
```

符合规范的代码示例如下：

```
FileReader in;
// Initialize stream
int inbuff;
char data;
while ((inbuff = in.read()) != -1) {
  data = (char) inbuff;
  // ...
}
```

第 5 章 PHP安全编码

随着互联网的发展,Web 2.0、云计算、物联网等热门概念不断催生新的产业和服务,与此同时,PHP 作为支撑这些新的产业和服务的技术体系也得到了空前发展。本章首先概述 PHP 开发的安全现状;其次对常见的 PHP 安全漏洞进行原理分析,提出可用的防范方案;最后总结 PHP 安全编码规范,以帮助程序设计人员更好、更安全地进行 Web 开发。

5.1 PHP 开发安全现状

互联网快速发展、不断创新的特点使得网站开发必须以最快的开发速度和最低的成本完成,才能保持领先性,以吸引更多的网民。而 PHP 的运行效率高、开发速度快、可扩展性强、开源自由等优点非常符合目前的互联网发展趋势,因此,越来越多的 Web 应用选择 PHP 作为主流的技术方案。

1. PHP的优点

PHP(Hypertext Preprocessor,超文本预处理器)是一种通用开源脚本语言。PHP 最初的英文全称是 Personal Home Page(个人主页),它是用 Perl 语言编写的,是用来显示个人履历以及统计网页流量的个人网页工具程序。后来,随着信息技术的发展,PHP 可以和数据库连接,生成动态网页程序,因此逐步应用于 Web 开发领域,在各种规模的网站中应用广泛。

与其他同类编程语言相比,PHP 有以下优点:

(1) 语法简单。其语法吸收了 Java、C 以及 Perl 等语言的优点,语法简单,初学者容易掌握。

(2) 开源自由。PHP 是免费的开源代码。它有很多较为成熟的资源,例如开源论坛 Discuz!、Phpwind 等,开源框架 Zend Framework、CakePHP、CodeIgniter、symfony 等,开源博客 WordPress 等,开源网店系统 Ecshop、ShopEx 等,开源 SNS 系统 UCHome、ThinkSNS 等,可充分满足使用者的各种应用需求。

(3) 效率高。由于 PHP 是将程序嵌入 HTML 文档中执行的,用 PHP 作出的动态页面的执行效率比完全生成 HTML 标记的 CGI 要高许多,而且 PHP 能实现 CGI 的所有功能。PHP 与一般的脚本代码不同,它可以执行编译后的代码,编译时可以对代码进行加密和优化,使代码运行更快、效率更高。

(4) 可扩展性强。PHP 可以用 C、C++ 等语言进行程序的扩展。

（5）跨平台能力强。PHP 的跨平台能力使它支持几乎所有流行的数据库以及操作系统。PHP 常与免费的开源 Web 服务器 Apache 和数据库 MySQL 配合，运行在 Linux 平台上（这个组合简称 LAMP，号称"Web 架构黄金组合"），具有很高的性价比，不仅能够降低使用成本，还能够提升开发速度，满足最新的互动式网络应用开发的需求。

（6）面向对象。PHP 4、PHP 5 在面向对象方面进行了很大的改进，现阶段的 PHP 已经可以用来开发大型商业程序。

PHP 适合快速开发中小型应用系统，能够对变动的需求做出快速反应。目前，在全球几千万个互联网网站中，有半数以上使用的是 PHP 技术。越来越多的公司对开发语言的选择从 ASP、Java 转到了 PHP，这种现象使得 PHP 社区越来越活跃，而活跃的 PHP 社区又反过来影响到很多项目或公司的开发语言选择，形成良性循环。PHP 的快速、开发成本低、周期短、后期维护费用低、开源产品丰富等优点都是其他语言所无法相比的。

2 PHP 开发的安全问题

随着越来越多的公司选择使用 PHP 进行网站等 Web 应用程序开发，PHP 开发的安全问题开始慢慢浮现。相对于其他语言，PHP 最大的问题就是语法不够严谨，缺乏统一的编码规范。例如，PHP 对函数名、方法名、类名是不区分大小写的，变量不需要定义就可以使用，虽然这些问题在编译时一般可以通过，但是在实际执行中容易发生不可控的问题。攻击者还可能利用这些编码漏洞进行注入攻击，窃取隐私信息，造成系统受损。最初 PHP 采用的是面向过程式编程，因此不同开发者的编程风格各异。虽然现在 PHP 已经支持面向对象，各类框架的命名规范也在一定程度上约束了编码人员的风格，但编码不规范的情况依然非常普遍，造成程序后期维护困难。而其他语言（如 Java）更为规范、严格，有利于维护和阅读，即使是初学者也可以写出规范的代码。

1）弱数据类型语言

PHP 是弱数据类型语言，也称为弱类型定义语言。与强类型定义语言相反，在 PHP 中，数据类型可以被忽略，即在定义一个变量的时候不需要为它指定数据类型，而是在解释的时候根据值的情况动态地赋予变量数据类型。弱数据类型语言不符合"所见即所得"的原则，定义的变量类型是不可预见并且可以改变的，变量类型的不可控性会导致在执行过程中出现大量的变量类型"隐形转换"。在开发人员不清楚"隐形转换"规则的情况下，极容易产生不可预知的运行结果，例如，难以通过编译发现代码缺陷，难以优化编译以提升代码性能，开发时难以做出正确的语言提示，以及容易出现注入漏洞等安全性问题。

2）异常捕获系统能力弱

PHP 的生命周期设计决定了它的异常处理功能使用得并不广泛，因而此功能也一直不够完善。Discuz!、DECMS、Ecshop 等系统的异常处理能力都很弱，一旦执行期间发生异常，就可能导致程序崩溃，这对于业务稳定性要求高的系统影响很大。攻击者只需通过简单的分布式攻击就可以使系统崩溃，影响系统的正常业务。

3）易遭受注入攻击

PHP 文件使用 require()函数引入或者包含外部文件。当 PHP 文件被执行时，外部文件的内容就将被包含进该 PHP 文件中，如果包含的外部文件发生错误，系统将抛出错

误提示，并且停止 PHP 文件的执行。使用 require()函数可以包含 URL 或文件名，当函数中存在动态包含文件的时候，攻击者可以利用远程文件使得程序访问并执行恶意文件，会造成一系列安全隐患。

PHP 的安全开发是一个艰巨的任务。以现在的安全技术来说，对于 PHP 安全开发问题的建议是：尽量依据编码规范进行安全编码，以减少可能出现的漏洞；对于数据安全要求高的业务应尽量避免使用 PHP 开发。

5.2 PHP 常见安全漏洞

5.2.1 会话攻击

1. 会话攻击的原理

会话攻击是攻击者最常用的攻击手段之一。当客户端的浏览器连接到服务器后，服务器会给该客户端用户生成一个独立的会话(session)，然后交与服务器进行管理和维护，相当于每个用户对应一个服务器的身份 ID，称为会话 ID(session ID)。当用户发送 HTTP 请求时，HTTP 头内将包含会话 ID 的值，服务器通过 HTTP 头的会话 ID 值判断用户的请求，这样可以省去在转换到不同页面进行身份验证时重复输入用户名和密码的麻烦。但这种方法在方便用户的同时却增加了很多安全隐患，如果攻击者能够获得某个用户在某个网站的会话 ID 和其他存储在会话内的重要变量，就可以在此会话 ID 的生命周期内使用该用户的权限以达到自己的某种目的。一个会话的生命周期是从用户通过浏览器连接到服务器后开始的，在用户关闭浏览器、从网站注销或者在 20min 内没有进行任何操作后结束。

与会话有关的攻击主要分为两种。

第一种是会话固定攻击(session fixation)。这种攻击方式的核心要点是让合法用户使用攻击者预先设定的会话 ID 来访问被攻击的应用程序，一旦用户的会话 ID 被成功固定，攻击者就可以通过此会话 ID 来冒充合法用户访问应用程序。

会话固定攻击的应用场景如下：

(1) 攻击者访问网站 http://www.abc.com，获取他自己的会话 ID，如 SID=123。

(2) 攻击者给目标用户发送链接，并带上自己的会话 ID，如 http://www.abc.com/?SID=123。

(3) 目标用户点击了 http://www.abc.com/?SID=123 后，像往常一样，输入自己的用户名、密码，登录到网站。

(4) 由于服务器的会话 ID 不改变，则现在攻击者点击 http://www.bank.com/?SID=123 时，他就拥有了目标用户的身份。

第二种常见的会话攻击是会话劫持。攻击者通过各种攻击手段获取用户的会话 ID，然后利用被攻击用户的身份登录相应网站。

攻击者获取用户的会话 ID 的方法有很多，下面是最常用的 3 种方法。第一种方法是

暴力获取，攻击者可以无数次猜测和尝试，直到蒙对会话 ID 为止，但由于一般 ID 都是随机产生的，这种方法耗时耗力，因此很少使用；第二种方法是通过计算获取，需要攻击者进行大量的逻辑分析和运算，因此这种方法也会耗费大量时间和精力；第三种方法是绝大多数攻击者使用的方法，即通过网络截获、安装病毒或者利用跨站脚本攻击获得用户的会话 ID。

2. 会话攻击的防范措施

对于会话攻击，通常用以下几种方法进行防范：

(1) 定期更换会话 ID，可以用 PHP 自带函数实现这一点。

(2) 更换会话名称。通常情况下会话的默认名称是 PHPSESSID，这个变量一般是在 Cookie 中保存的。如果更改了会话的名称，攻击者将不容易找到会话的存储地点，就可以阻止攻击者的一部分攻击。

(3) 对透明化的会话 ID 进行关闭处理。透明化是指在 HTTP 请求中没有使用 Cookie 来指定会话 ID 时，会话 ID 使用链接来传递。关闭透明化会话 ID 可以通过修改 PHP. ini 文件中的相关配置来实现。

(4) 通过 URL 传递隐藏参数。这样，即使攻击者获取了会话数据，但是由于相关参数是隐藏的，攻击者也很难获得会话 ID。

(5) 设置 HttpOnly。将 Cookie 的 HttpOnly 设置为 true，可以防止客户端脚本访问 Cookie，从而有效地防止 XSS 攻击。

5.2.2　命令注入攻击

1. 命令注入攻击原理

命令注入(command injection)漏洞是 PHP 应用程序中常见的脚本漏洞之一，是指攻击者可以通过构造特殊命令字符串的方式将数据提交至 Web 应用程序中，并利用该方式执行外部程序或系统命令以实施攻击，非法获取数据或者网络资源等。命令注入漏洞的存在十分广泛，国内许多著名的 PHP 应用程序，如 Discuz!、dedecms 等均被发现存在命令注入攻击漏洞。攻击者可以通过命令注入攻击漏洞快速获取网站权限，进而实施挂马、钓鱼等恶意攻击，造成巨大的影响和危害。

命令注入攻击最初被称为 Shell 命令注入攻击，是由挪威一名程序员在 1997 年意外发现的，他通过构造命令字符串的方式从一个网站删除了网页，进行了命令注入攻击。使用系统命令是一项危险的操作，尤其在程序设计人员试图使用远程数据来构造要执行的命令时，如果使用了被污染的远程数据，命令注入漏洞就产生了。命令注入漏洞存在的主要原因是程序设计人员在应用 PHP 语言中一些具有命令执行功能的函数时，对用户提交的数据内容没有进行严格的过滤就带入函数中执行而造成的。例如，当攻击者提交的数据内容为向网站目录写入 PHP 文件时，就可以通过该命令注入攻击漏洞写入一个 PHP 后门文件，进而实施进一步的渗透攻击。

在 PHP 中，可以用以下 4 个命令执行函数和一个运算符执行外部程序或函数：

(1) system()。该函数可以用来执行一个外部的应用程序并将相应的执行结果输

出，函数原型如下：

```
string system(string comand, int &return_var)
```

其中，command 是要执行的命令，return_var 存放执行命令后的状态值。

(2) exec()。该函数可以用来执行一个外部的应用程序，函数原型如下：

```
string exec(string comand, array &output, int &return_var)
```

其中，command 是要执行的命令，output 存放命令输出的每一行字符串，return_var 存放执行命令后的状态值。

(3) passthru()。该函数可以用来执行一个 UNIX 系统命令并显示原始的输出，当 UNIX 系统命令的输出是二进制的数据，并且需要直接返回值给浏览器时，需要使用 passthru()函数来代替 system()与 exec()函数。passthru()函数原型如下：

```
void passthru(string comand, int &return_var)
```

其中，command 是要执行的命令，return_var 存放执行命令后的状态值。

(4) shell_exec()。该函数执行 shell 命令并返回输出的字符串，函数原型如下：

```
string shell_exec(string comand)
```

其中，command 是要执行的命令。

(5) “`”运算符。与 shell_exec()函数的功能相同，执行 shell 命令并返回输出的字符串。

2. 命令注入攻击的防范方法

由于目前 PHP 语言广泛应用于 Web 应用程序开发，Web 应用程序设计人员需要了解命令注入攻击漏洞的危害，修补程序中可能存在的被攻击者利用的漏洞，保护网络用户的安全，使网站免受挂马、钓鱼等恶意代码的攻击。通常程序设计人员可以通过以下几种方法防范命令注入攻击：

(1) 尽量不要执行外部的应用程序或命令。

(2) 使用自定义函数或函数库实现外部应用程序或命令的功能。

(3) 在执行 system()等命令执行函数前，要确定参数内容。

(4) 使用 escapeshellarg()函数处理相关参数。该函数会将任何引起参数或命令结束的字符进行转义，如单引号“”会被转义为“\”，双引号“"”会被转义为“\"”，分号“;”会被转义为“\;”，这样 escapeshellarg()函数会将参数内容限制在一对单引号或双引号中，转义后的单引号或双引号无法截断当前命令的执行，达到了防范命令注入攻击的目的。

5.2.3 客户端脚本注入攻击

1. 客户端脚本注入攻击原理

客户端脚本注入攻击是指攻击者将可执行的脚本插入表单、图片、动画或超链接文字等对象内，当用户留言或者打开这些对象后，浏览器将执行攻击者所植入的恶意脚本，通过使用户自动跳转到一些攻击者恶意构造的网站等行为实施攻击。此漏洞在以前的

PHP 网站中经常存在，但随着 PHP 版本的升级，这些问题已经慢慢减少。

可以用于脚本植入的 HTML 标签一般包括以下几种：

(1) <script>标签标记的 JavaScript 和 VBScript 等页面脚本程序。在<script>标签内可以指定 JavaScript 程序代码，也可以在 src 属性内指定 JavaScript 文件的 URL 路径。

(2) <object>标签标记的对象。这些对象是 Java Applet、多媒体文件和 ActiveX 控件等。通常在 data 属性内指定对象的 URL 路径。

(3) <embed>标签标记的对象。这些对象是多媒体文件，例如 swf 文件。通常在 src 属性内指定对象的 URL 路径。

(4) <applet>标签标记的对象。这些对象是 Java Applet，通常在 codebase 属性内指定对象的 URL 路径。

(5) <form>标签标记的对象。通常在 action 属性内指定要处理表单数据的 Web 应用程序的 URL 路径。

客户端脚本注入攻击的步骤如图 5-1 所示。具体如下：

(1) 攻击者在注册成普通用户后，登录网站。

(2) 打开留言页面，插入用来进行攻击的 JavaScript 代码。

(3) 其他用户(包括管理员)登录网站，浏览此留言的内容。

(4) 隐藏在留言内容中的 JavaScript 代码被执行，攻击被发动。

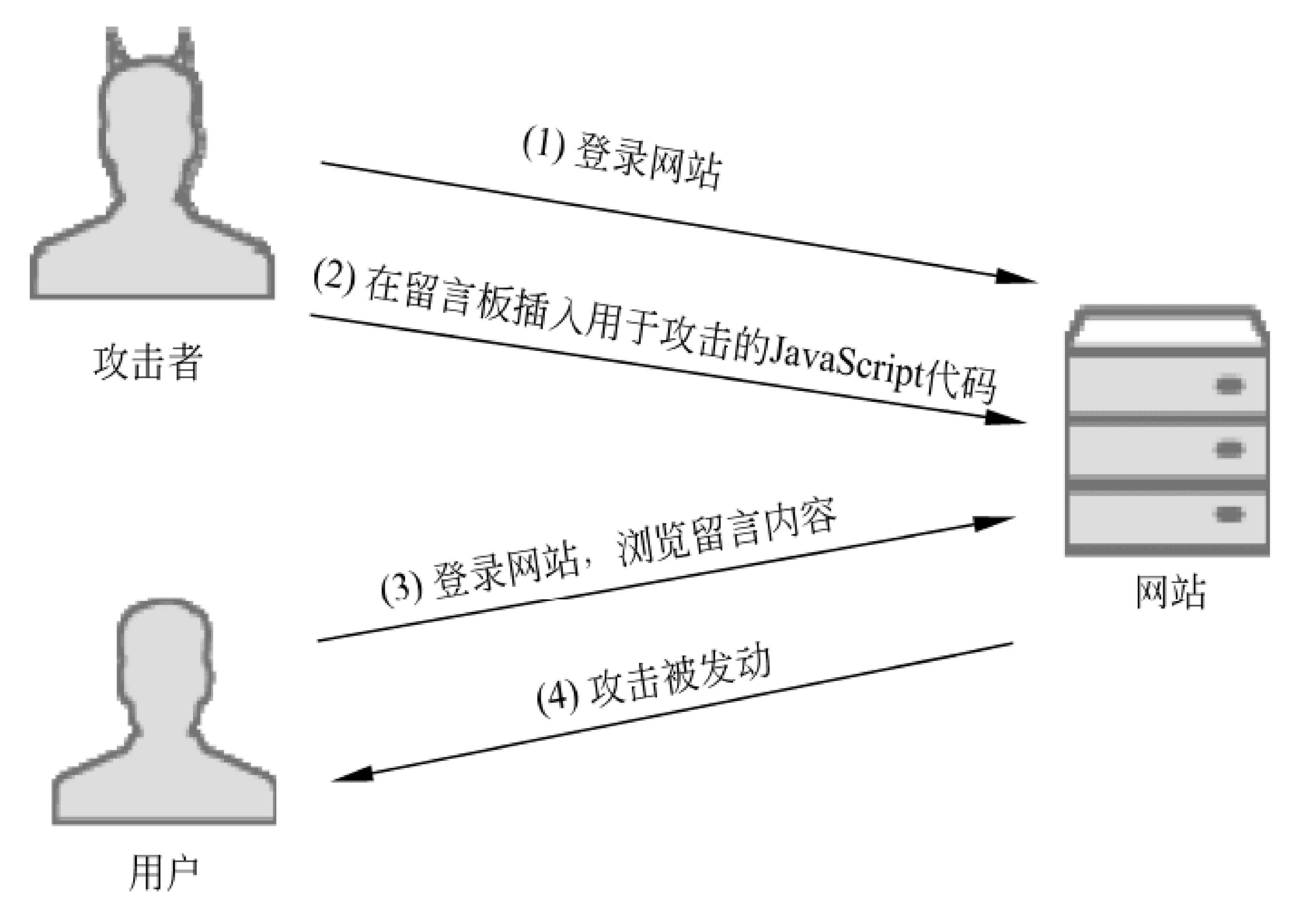


图 5-1　客户端脚本注入攻击步骤

2 客户端脚本注入攻击的防范方法

攻击者利用客户端脚本注入漏洞进行攻击的手段是灵活多变的，因此需要采取多种防范方法，才能有效防止攻击者对脚本注入漏洞进行攻击。常用的方法如下：

(1) 对可执行文件的路径进行预先设定。可以通过设置安全模式 safe_moade_exec_dir 来实现。如果 PHP 使用了安全模式，system()和其他程序执行函数将拒绝启动不在此目录中的程序，必须使用/作为目录分隔符。

(2) 对命令参数进行处理，一般使用 htmlspecialchars()函数来将特殊字符转换成 HTML 编码。

(3) 用系统自带的函数库代替外部命令。

(4) 进行系统操作的时候尽可能不使用外部命令。

5.2.4 变量覆盖漏洞

1. 常见的变量覆盖漏洞

变量覆盖漏洞是指用传入的参数值替换程序原有的变量值的情况。一般变量覆盖漏洞需要结合程序的其他功能实现完整的攻击。经常导致变量覆盖漏洞的场景有：全局变量注册未关闭，$ $遍历初始化变量，extract()函数、parse_str()函数和 import_request_variables()函数使用不当，等等。

1) 全局变量漏洞

PHP 中的全局变量不像其他开发语言一样需要预先声明，而是可以直接使用，PHP 会在第一次使用时自动创建变量，并根据上下文环境自动确定变量的类型。这对于程序设计人员而言是十分方便的，因为只要一个变量被创建，就可以在程序中的任何地方使用。但 PHP 的这种特性也导致程序员在编写 PHP 程序的过程中很少初始化变量，通常直接使用变量被创建时默认的空值，这使得攻击者可以通过给全局变量赋值来欺骗代码，达到其恶意的目的。

register_globals 是 PHP 中的一个控制选项，其意思是注册为全局变量。当其状态是 on(开启)的时候，传递过来的值会被直接注册为全局变量，如果该变量在此之前有值，这个值会被覆盖，因此变量的 register_globals 选项应时刻保持 off(关闭)状态，以防止攻击者恶意传递参数值，覆盖系统变量。register_globals 选项在 PHP 4 及以前版本默认开启，在 PHP 5 以后默认关闭。

2) $$遍历初始化变量问题

使用 foreach 来遍历数组中的值时，会将获取的数组的键名作为变量，将数组的键值作为变量的值，因此就产生了变量覆盖漏洞。请求?id=1 会将 $ id 的值覆盖，使得 $ id=1。

3) extract()变量覆盖

extract()函数从数组中将变量导入当前的符号表，使用数组键名作为变量名，使用数组的键值作为变量值，针对数组中的每个元素在当前符号表中创建一个对应的变量。该函数在以下 3 种情况下会覆盖已有变量的值：

(1) 当第二个参数为 EXTR_OVERWRITE 时。这表示如果有冲突，则覆盖已有的变量的值。

(2) 当只传入第一个参数时。这时候默认为 EXTR_OVERWRITE 模式。

(3) 当第二个参数为 EXTR_IF_EXISTS 时。这表示仅在当前符号表中存在同名变量时覆盖它们的值。

4) parse_str()变量覆盖

parse_str()函数的作用是解析字符串并将其注册为变量。parse_str()函数有两个参数：

第一个参数代表要解析并注册为变量的字符串;第二个参数是一个数组,注册的变量会放到这个数组里。如果没有第二个参数,则由该函数设置的变量将覆盖已存在的同名变量。

5）import_request_variables()变量覆盖

import_request_variables()函数的作用是把 GET、POST 和 COOKIE 的参数注册为变量,该功能用在 register_globals 为 off 的时候。不过,这个函数在 PHP 5.4 之后就被取消了。

import_request_variables()函数有两个参数：第一个参数代表要注册的变量,其中 G、P、C 分别代表 GET、POST、COOKIE;第二个参数为要注册的变量前缀。import_request_variables()函数可以在 register_global 为 off 时将 GET、POST、COOKIE 变量导入全局作用域中。

2 变量覆盖漏洞的防范方法

对于 PHP 的变量覆盖漏洞问题,通常采用以下的防范方法：

(1) 使用原始变量数组。建议直接用原始的变量数组(如 $_POST、$_GET 等)进行操作,避免使用变量注册功能。

(2) 验证变量是否存在。注册变量前,先判断变量是否存在。使用 extract()函数时可以配置第二个参数为 EXTR_SKIP。自行创建的变量一定要初始化。

5.2.5　危险函数

为了系统的安全起见,常在程序设计时禁用一些 PHP 危险函数。具体如下：

```
phpinfo()
```

功能描述：输出 PHP 环境信息以及相关的模块、Web 环境等信息。

危险等级：中。

```
passthru()
```

功能描述：允许执行一个外部程序并回显输出,类似于 exec()。

危险等级：高。

```
exec()
```

功能描述：允许执行一个外部程序(如 UNIX Shell 或 CMD 命令等)。

危险等级：高。

```
system()
```

功能描述：允许执行一个外部程序并回显输出,类似于 passthru()。

危险等级：高。

```
chroot()
```

功能描述：可改变当前 PHP 进程的工作根目录,仅当系统支持 CLI 模式的 PHP 时才能工作,且该函数不适用于 Windows 系统。

危险等级：高。

scandir()

功能描述：列出指定路径中的文件和目录。
危险等级：中。

chgrp()

功能描述：改变文件或目录所属的用户组。
危险等级：高。

chown()

功能描述：改变文件或目录的所有者。
危险等级：高。

shell_exec()

功能描述：通过 shell 执行命令，并将执行结果作为字符串返回。
危险等级：高。

proc_open()

功能描述：执行一个命令并打开文件指针用于读取以及写入。
危险等级：高。

proc_get_status()

功能描述：获取使用 proc_open()打开的进程的信息。
危险等级：高。

error_log()

功能描述：将错误信息发送到指定位置(文件)。
安全备注：在某些版本的 PHP 中，可使用 error_log()绕过 PHP 安全模式，执行任意命令。
危险等级：低。

ini_alter()

功能描述：它是 ini_set()函数的一个别名函数，两者功能相同。具体参见 ini_set()。
危险等级：高。

ini_set()

功能描述：可用于修改、设置 PHP 环境配置参数。
危险等级：高。

ini_restore()

功能描述：可用于将 PHP 环境配置参数恢复为初始值。
危险等级：高。

dl()

功能描述：在 PHP 运行过程中(而非启动时)加载 PHP 外部模块。
危险等级：高。

pfsockopen()

功能描述：建立一个 Internet 或 UNIX 域的 socket 持久连接。
危险等级：高。

syslog()

功能描述：可调用 UNIX 系统的系统层 syslog()函数。
危险等级：中。

readlink()

功能描述：返回符号连接指向的目标文件内容。
危险等级：中。

symlink()

功能描述：在 UNIX 系统中建立一个符号链接。
危险等级：高。

popen()

功能描述：可通过该函数的参数传递一条命令，并执行该函数打开的文件。
危险等级：高。

stream_socket_server()

功能描述：建立一个 Internet 或 UNIX 服务器连接。
危险等级：中。

putenv()

功能描述：用于在 PHP 运行时改变系统字符集环境。在低于 5.2.6 版本的 PHP 中，可利用该函数修改系统字符集环境，然后利用 sendmail 指令发送特殊参数，执行系统 shell 命令。
危险等级：高。

5.3 PHP 安全编码规范

当一个软件项目尝试制定一致的编码规范时，可以使参与项目的开发人员更容易了解项目中的代码，弄清程序的状况，使新的参与者可以很快地适应环境，防止部分参与者为了节省时间而自创一套风格，导致其他项目人员在阅读程序时浪费过多的时间和精力。

而且在编码规范一致的环境下，也会减少编码出错的机会。虽然由于每个人的编码习惯不同，可能需要一段时间来适应和改变自己的编码风格，暂时降低了工作效率，但从项目长远健康的发展以及后期更高的团队工作效率来考虑，暂时的工作效率降低是值得的，也是必须经过的一个过程。编码规范虽然并不是项目成功的关键，但可以帮助项目人员在团队协作中有更高的效率，更加顺利地完成既定任务。

制定统一的编码规范对于项目开发来说非常重要。下面是PHP开发中常用的编码规范。熟悉并遵守这些编码规范，不但可以使程序员养成良好的开发习惯，增强程序的可读性、可移植性和可重用性，还能提高项目开发效率。

5.3.1 语言规范

1. 控制结构使用规范

对于控制结构if-else、switch、for、while等的使用应遵循以下规范：

(1) 在if条件判断中，如果用到常量判断条件，将常量放在等号或不等号的左边，例如：if (6== $errorNum)，这样，当程序员在等式中漏了一个等号时，语法检查器将会报错，可以帮助程序员很快找到错误位置。

(2) 在switch结构中必须有default块。

(3) 在for和while的循环中，要仔细检查continue、break语句的使用，避免产生类似goto语句的问题。

2. 类的构造函数

如果要在类中编写构造函数，必须遵循以下规范：

(1) 不能在构造函数中加入太多实际操作，构造函数应主要用来初始化一些值和变量。

(2) 不能在构造函数中因为实际操作而返回false或者错误，这是因为在声明和实例化一个对象的时候是不能返回错误的。

3. true/false和0/1判断的书写规范

(1) 不能使用0/1代替true/false，因为在PHP中这两种形式是不同的。

(2) 不要使用非零的表达式、变量或者方法直接进行true/false判断，而必须使用严格的、完整的true/false判断。例如，不要使用if ($a)或者if (check())，而要使用if (false != $a)或者if (false != check())。

4. 避免嵌入式赋值

在程序中避免下面例子中的嵌入式赋值：

```
while ($a != ($c = getchar()))
{
…      //process the character
}
```

5. 错误处理规范

对于系统错误信息，应注意以下两点：

(1) 检查所有系统调用的错误信息，除非想要忽略错误。

(2) 为每条系统错误信息定义系统错误文本，并记录错误日志。

5.3.2　程序注释

每个程序都应该提供必要的注释。书写注释要求规范严谨，为今后利用 phpDoc 生成 PHP 文档做准备。与此同时，规范严谨的注释能够提高代码审查的效率，有利于安全漏洞的发现和修复。程序注释的总体原则如下：

(1) 注释中除了文件头的注释块外，其他地方都不使用//注释，而使用/*　*/注释。

(2) 注释内容必须写在被注释对象的前面，而不应写在一行中或者一行的后面。

1. 程序头注释块注释规范

在每个程序的头部必须有统一的注释块，其注释规范如下：

(1) 必须包含本程序的描述。

(2) 必须包含作者。

(3) 必须包含编写日期。

(4) 必须包含版本信息。

(5) 必须包含项目名称。

(6) 必须包含文件的名称。

(7) 重要的使用说明，如类的调用方法、注意事项等。

参考例子如下：

```
<?php
//
// +----------------------------------------------------------+
// | PHP version 4.0                                          |
// +----------------------------------------------------------+
// | Copyright (c) 1997-2001 The PHP Group                    |
// +----------------------------------------------------------+
// | This source file is subject to  of the PHP license,      |
// | that is bundled with this packafile LICENSE, and is      |
// | available at through the world-web at                    |
// | http://www.php.net/license/2_02.txt.                     |
// | If you did not receive a copy of the  and are unable to  |
// | obtain it through the world-wide-web,end a note to       |
// | license@php.net so we can mail you a immediately.        |
// +----------------------------------------------------------+
// | Authors: Stig Bakken <ssb@fast.no>                       |
// |          Tomas V.V.Cox <cox@idecnet.com>                 |
// |                                                          |
// +----------------------------------------------------------+
//
// $Id: Common.php,v 1.8.2.3 2001/11/13 01:26:48 ssb Exp $
```

2. 类的注释

类的注释应写在类的前面，采用下面的例子中的注释方式：

```
/**
 * @Purpose:
 * 执行一次查询
 * @Method Name: Query()
 *
 * @Param: string $queryStr SQL查询字符串
 * @Param: string $username 用户名
 *
 * @Author: Michael Lee
 *
 * @Return: mixed 查询返回值（结果集对象）
 */
function ($queryStr,$username)
{……}
```

3. 函数和方法的注释

函数和方法的注释应写在函数和方法的前面，采用下面的例子中的注释方式：

```
/**
 * @Purpose:
 * 执行一次查询
 * @Method Name: Query()
 *
 * @Param: string $queryStr SQL查询字符串
 * @Param: string $username 用户名
 *
 * @Author: Michael Lee
 *
 * @Return: mixed 查询返回值（结果集对象）
 */
function ($queryStr,$username)
{……}
```

4. 变量或者语句注释规范

(1) 变量或者语句注释写在上面一行，而不写在变量或者语句的同一行或者该行的后面。

(2) 注释采用/* */的方式。

(3) 每个函数前面要包含一个注释块，内容包括函数功能简述、输入输出参数、预期的返回值和出错代码定义。

(4) 注释应完整、规范。

(5) 把已经注释掉的代码删除，或者注明这些已经注释掉的代码仍然保留在源码中的特殊原因。

变量注释的例子如下：

```
/**
* @Purpose:
* 数据库连接用户名
* @Attribute/Variable Name: db_user_name
* @Type: string
*/
var db_user_name;
```

5.3.3 项目规范

1. PHP项目通常的文件目录结构

建议在开发规范的、独立的 PHP 项目时，使用规范的文件目录结构，这有助于提高

项目的逻辑结构合理性，对于扩展和合作以及团队开发均有好处。

一个完整、独立的 PHP 项目通常的文件和目录结构如下：

/：项目根目录。

/manage：后台管理文件存放目录。

/css：CSS 文件存放目录。

/doc：存放项目文档。

/images：所有图片文件存放目录(在其中根据需要设立子目录)。

/scripts：客户端 JavaScript 脚本存放目录。

/tpl：网站所有 HTML 的模板文件存放目录。

/error.php：错误处理文件(可以定义到 Apache 的错误处理中)。

以上是 PHP 项目通常的目录结构，根据具体应用的实际情况，不一定完全遵循这个目录结构，但是应尽量做到规范化。

2. PHP 和 HTML 代码的分离问题

对性能要求不是很高的项目和应用，建议不采用 PHP 和 HTML 代码混排的方式书写代码，而采用 PHP 和 HTML 代码分离的方式，即采用模板的方式处理。这样不仅使程序逻辑结构更加清晰，也有助于开发过程中人员的分工安排，同时还对日后项目的页面升级改版提供了更多便利。

对于一些特殊情况，例如对性能要求很高的应用，可以不采用模板方式。

3. PHP 项目开发中的程序逻辑结构

对于 PHP 项目开发中的程序逻辑结构，应遵循以下规范：

(1) 对于 PHP 项目开发，尽量采用面向对象的思想开发。PHP 5 及以后版本在面向对象的开发功能方面大有提高。

(2) 在 PHP 项目中，建议将独立的功能模块尽量写成函数调用。对于一整块业务逻辑，建议封装成类，既可以提高代码可读性，也可以提高代码重用性。例如，通常将对数据库的接口封装成数据库类，有利于平台的移植。

(3) 重复的代码要做成公共的库(插件产品除外。插件产品系列有多个相似的产品，为了尽可能地减小安装包尺寸，不适合将这些产品共用的所有函数做成公共的库)。

5.3.4 特殊规范

1. 变量定义

某特定开发环境下的 PHP 代码编写规范要求所有的变量均需要先声明后使用，否则会有错误信息。对于数组，在使用一个不确定的 key 时，应先进行 isset()的判断，然后再使用，例如：

```
$array = array();
$var = isset($array[3]) ? $array[3] : "";
```

2. 引用的使用

引用在程序中使用得比较多。引用共用同一个内存区域，而不需要另外进行复制。某特定开发环境下使用引用时，需要注意下面的情况。

在函数的输入参数中使用引用时，不能在调用的时候在输入参数前加 & 来引用，直接使用该变量即可，同时必须在函数定义的时候说明输入参数来自引用，例如下面的代码：

```
$a = 1;
function ab(&$var)
{
$var ++;
return $var;
}
$b = ab($a) // 注意，此处不能使用 $b = ab(&$a)的方式;
echo $b.”/n”;
echo $a.”/n”;
```

此时 $a 和 $b 都是 2。

某特定开发环境下对引用的特殊要求源自 php.ini 文件中 allow_call_time_pass_reference 项设置，其对外公开的版本是 on，这样就可以支持在调用函数时将 & 直接加到变量前面进行引用，但是这一方法在将来版本的 PHP/Zend 里可能不再支持，因此程序员最好关闭这一选项(使用 off，在此特定的所有运行环境下都是 off)，并确认脚本仍能正常工作，以保证在将来版本的 PHP 中它们仍能工作。

3. 变量的输入输出

在某特定开发环境下，对 Web 应用程序通过 GET 或者 POST 方法传递来的参数均要求进行严格的过滤和合法性验证，不推荐使用 $_GET、$_POST 或者 $_REQUEST 直接获取，而应通过此特定环境的_yiv 模块提供的方法获取和过滤处理。

5.3.5 配置安全

由于配置不当引发的安全问题屡见不鲜。通过一系列的安全配置，可以有效地解决一些安全隐患，从而为系统增加安全系数。

1. 关闭注册全局变量功能

register_globals 的功能是注册全局变量，on 是开启，off 是关闭。在开启该功能后会带来巨大的安全隐患，所以建议关闭该功能。register_globals 在 PHP 4.2.0 后默认为 off。如果根据需求需要开启该功能，可使用两个方法进行防御：一个是初始化变量，另一个是配置最高预警信息。

安全隐患举例：

```
<?php
if(!empty($_COOKIE[' secret ']))
    { $authorized = true; }
    if($authorized){
        …//do some authoration action
    }
?>
```

在上面的代码中，若没有通过 Cookie 认证，那么 $authorized 将一直为假，就无法被认证。但是如果 register_globals 为 on，那么在 URL 中就可以修改 get 参数，将 $authorized 注册为全局变量，并修改它的值为真，例如 http://xxx/test.php?authorized=1，这样就能够绕过 Cookie 认证。

2. 配置错误信息

在攻击者的渗透过程中，系统错误信息的暴露给攻击者提供了有利条件。所以，在开发过程中，可开启将错误信息输出到浏览器的设置；但是在程序上线后，需关闭错误信息提示。可以关闭浏览器显示错误提示，将错误提示记录到本地日志中。这些可以在 php.ini 中设置，也可以在 PHP 程序中设置。

3. 关闭 allow_url_include 和 allow_url_fopen 功能

allow_url_include 允许加载远程 PHP 文件，allow_url_fopen 允许加载远程本地写文件。这两个功能开启后带来的安全隐患也是巨大的，会带来远程文件包含漏洞，所以建议将其关闭。如果有需求，应对外来变量进行过滤。

第 6 章 Python安全编码

随着大数据和云计算的发展，Python因在开发大型和复杂应用程序方面的便捷性而大受欢迎。作为最受欢迎的程序设计语言之一，Python语法简洁，代码执行快速。随着Python的广泛应用，Python的安全编程规范也越来越受到开发人员的重视。本章首先介绍Python开发的安全现状；其次对常见的安全漏洞进行分析，并提出防范措施；最后介绍Python安全编码规范，以指导开发人员进行安全编码。

6.1 Python 开发安全现状

Python是一种面向对象的解释型计算机程序设计语言，也是一种功能强大的通用型语言，它是由荷兰人Guido van Rossum于1989年发明的。Python语法简洁清晰，设计者很少强调传统的语法，因此无论对于开发人员还是普通用户来说都很容易使用。

Python是一个面向企业和项目的、基于生产的语言，其应用程序可以在绝大多数环境中使用和部署，并且不会随平台变化而损失性能。Python具有丰富和强大的库，可满足多种开发需求，因而被逐渐广泛应用于各种领域，例如系统管理任务的处理、Web应用开发、桌面应用程序开发、服务器部署、科学建模等。开发人员在深入了解Python之后，就能具备可以适应范围更宽的工作角色的技能。在IEEE发布的2017年编程语言排行榜中，Python高居首位。

1. Python开发的优点

Python开发近些年如此热门的原因有以下几点。

1）高度的可阅读性

作为一种解释型语言，Python更强调代码的可读性和语法的简洁。Python尽量使用其他语言经常使用的标点符号和英文单词，让代码看起来整洁美观。相比C++或Java等语言，Python能让开发者用更少的代码来表达想法。它不像其他的静态语言（如C、Pascal）那样需要重复书写声明语句，也不像它们的语法那样经常有特殊情况和意外发生。不管是小型程序还是大型程序，Python程序的结构都清晰明了。Python的这种特性使程序设计人员能够专注于解决问题而非表现语言的丰富程度。采用强制缩进的方式使得代码具有极佳的可读性。

2）简单易学

在众多的编程语言中，Python是最容易理解和学习的编程语言之一，这是由于它在设计的时候对语法进行了简化并强调了自然语言风格。Python虽然是用C语言编写的，

但是它摈弃了 C 语言中复杂的指针，这样，程序设计人员就能够更快地编写 Python 代码并执行它。无论是对于新手还是对经验丰富或已经熟悉 Java、C 或者 Perl 的开发人员来说，Python 都十分简单易学。

3）丰富全面的库

当程序设计人员需要开发庞大的项目时，全面强大的库资源可以帮助程序设计人员节省时间和缩短开发周期。Python 以 PyPI 为其后盾，有丰富的标准库和可定义的第三方库，开发人员想从事任何方向的技术编程，几乎都能在 Python 中找到相应的库支持。这些库中的模块可以用来进行数据库处理、计算机视觉实现、维度分析等高级数据分析，或者构建 REST 风格的 Web 服务。它可以帮助程序设计人员处理各种工作，包括正则表达式、文档生成、单元测试、线程、数据库、网页浏览器、CGI、FTP、电子邮件、XML、XML-RPC、HTML、WAV 文件、密码系统、GUI（图形用户界面）和其他与系统有关的操作。

4）开源性

Python 的代码均是开源的，其源代码和解释器 CPython 遵循 GPL（GNU General Public License，GNU 通用公共许可）协议。开发人员可以自由地复制、阅读 Python 源代码，对其进行改动，或将源代码的一部分用于开发的新软件中。其中一些优秀的开源框架对 Python 发展的贡献很大，例如，用于 Web 开发 Python Web 框架 Django，用于网络编程的支持高并发的网络框架 Twisted，用于爬虫领域的 Scrapy、Request、BeautifulSoap、urllib 等框架，用于云计算的 OpenStack 等框架。

5）可跨平台

在计算机内部，Python 解释器把源代码转换成为字节码的中间形式，然后再把它翻译成计算机使用的机器语言并运行。由于 Python 虚拟机可以在绝大多数操作系统中运行，因此，只要程序员在编写程序时注意避免使用过于依赖于系统的特性，Python 程序就可以无须修改地工作在不同平台上，例如 Linux、Windows、MacOS、Macintosh 以及 Google 公司基于 Linux 开发的 Android 平台等。此外，Python 强大的库资源也能很好地提供跨平台的支持。

6）可嵌入性

当开发人员希望程序中的一段关键代码运行得更快时，可以把这部分程序用 C 或 C++ 编写，然后在 Python 程序中使用它们，这使得 Python 程序更加高效。Python 常同一些编程语言集成在一起。例如，CPython 是 Python 与 C 语言集成的版本，Jython 是 Python 与 Java 集成的版本，RubyPython 是 Python 与 Ruby 集成的版本等。

7）用户活跃度高

从 Python 诞生起到现在时间已经很久了，关于 Python 开发有大量的文档、指南、教程和开发人员支持，因此 Python 的开发者社区非常活跃，无论是新手程序员还是熟练的开发人员，在需要支持或有问题时，都能在社区中得到及时的帮助。此外，各大公司的支持对 Python 编程语言的发展也很有帮助。正如 C ♯ 有微软公司，Java 有 Sun 公司，Facebook 公司使用 PHP 一样，谷歌公司在近年大量使用 Python，并将其应用在多个平台和应用中。为了方便公司团队和未来的开发人员使用系统和应用程序，谷歌公司编写了大量的 Python 指南和教程，提供了越来越多的文档和支持工具。

2. Python的应用方向

近年来在网页爬虫、数据挖掘、科学计算、机器学习等领域，Python已经拥有不可替代的地位。Python的开发效率极高，可以帮助企业在短时间内将想法用产品表现出来，因此它已成为众多创业公司的首选开发语言。Python的应用方向主要有以下几个。

1）大数据和云计算方案

Python是数据科学中最流行的语言之一，仅次于R语言。由于Python拥有大量的标准库和扩展库，可以进行强大的数据操作，通过数据分析，充分挖掘数据的价值。Python在企业中常被用于机器学习和人工智能系统以及各种现代技术，已成为大数据分析的强有力工具。

2）自动化运维

目前Python是金融分析、量化交易领域里用得最多的科学计算语言，随着NumPy、SciPy、Matplotlib、Enthought librarys等众多程序库的开发，Python越来越适用于科学计算以及高质量的2D和3D图像绘制。与科学计算领域最流行的商业软件Matlab相比，Python更加通用，比Matlab所采用的脚本语言的应用范围更广泛。

3）网络游戏开发

Python在网络游戏开发中也有很多应用。相比Lua和C++，Python有更高阶的抽象能力，可以用更少的代码描述游戏业务逻辑。Python非常适合编写1万行以上的项目，而且可以很好地把网络游戏项目的规模控制在10万行代码以内。

3. Python的安全现状

Python的开源性使其应用广泛，但Python语言是开源的，一些攻击者可以利用Python的开源性发布含有漏洞的开源代码，当程序员直接使用这些代码而不对其进行安全检验时，软件就可能被攻击者利用。此外，相比其他语言，Python中的安全错误不易被发现。例如，当需要格式化字符串时，在PHP中需要借用sprintf函数，而在Python中只需要一个%即可，所以很多在PHP中可以轻易发现的错误在Python中就需要更加细心的检查。

1）开源软件爆炸式增长

包管理工具索引的开源包数量呈爆炸式增长，Python库一年的开源包数量增长率为32%。但是，开源代码具有内在风险。虽然向开源项目贡献代码的多数人员并非恶意人员，但是他们使用的开源代码的来源无法保证，而且不同开源项目之间的安全标准和维护者的技术水平各不相同。开源组件中的已知漏洞也成为非常具有吸引力的攻击向量(attack vector)，一旦这些漏洞被公开披露，那么利用代码就会紧随其后。

2）模块注入

Python的模块导入体系很强大，同时也很复杂。调用Python解释器的最初脚本所在的目录是自动插进搜索路径的，任何模块和包都可以通过其文件名或目录名在被sys.path列表所定义的搜索路径中找到。搜索路径初始化是一个复杂的过程，而且也会因为Python版本、平台和本地配置的不同而有所差异。因此，攻击者只要能找到一个方法在Python导入模块时将恶意的模块插进Python可能会导入的目录或者包文件中，就能成

功地制造一次对 Python 应用的攻击。

3）临时文件的使用

临时文件的不正当使用会导致一系列的安全漏洞，这对很多编程语言都是一大威胁。在 Python 脚本中，临时文件仍然在被大量使用，临时文件一般被创建和保存在文件系统中而非开发者最终要将其放置的位置。攻击者可以利用不安全的文件系统的访问许可，以此来调用一些中间步骤，最终引发数据保密和完整性问题。

6.2 Python 常见安全威胁的防御

6.2.1 代码注入的防御

解释语言与编译语言不同，因编译语言需要程序员在编写时就对源代码进行编译并生成可执行文件，但解释语言在执行之前才生成机器代码。在编译语言中，一旦程序被编译，除非重新编译并重新分发可执行文件，否则源代码将无法再被修改，而现在的 Web 应用程序都需要使用后端数据库来收集数据并为用户生成动态驱动的内容，编译语言已经不适用于目前的应用开发中，因此现阶段解释型编程语言在开发人员中更加流行，包括 PHP、Java、Python 等。

1. 代码注入的原理

代码注入攻击指的是任何允许攻击者在网络应用程序中注入源代码并使其得到解读和执行的方法。源代码可以通过不可信的输入直接注入，也可以在本地文件系统或类似于 URL 的外部来源加载代码时操纵网络应用程序。造成代码注入漏洞的原因主要有以下几个：输入验证失败，任何语境下的不可信输入，未能保障源代码库的安全，在下载第三方程序库时不够谨慎，服务器配置不当导致非 Python 文件可以通过网络服务器传送到 Python 解释程序，这意味着不可信用户可以上传文件到服务器，从而带来极大的风险。

Python 中常见的代码注入攻击方法如下：

(1) eval()函数。该函数能够执行一行任意字符串形式的代码，将字符串对象转换为有效的表达式，参与求值运算并返回计算结果。其格式如下：

```
eval(expression,globals=None, locals=NoneD)
```

其中，expression 是一个参与计算的 Python 表达式；globals 是可选的参数，代表了全局名称空间中的对象，如果其属性不是 None，就必须是 dictionary；locals 是可选的对象，代表了局部名称空间中的对象，如果其属性不是 None，可以是任意的映射对象。

eval()函数需要用户输入一个表达式。如果攻击者输入恶意字符串，例如：

```
__import__('os').system('dir')
```

则当前目录文件都会展现在攻击者前面，攻击者可以继续输入命令查看系统文件，盗取敏感信息。

（2）exec()函数。该函数能够执行字符串形式的代码块。其格式如下：

```
exec obj
```

其中，obj 对象可以是字符串（如单一语句、语句块），文件对象，也可以是已经由 compile 预编译过的代码对象。

```
exec("__import__('os').system('uname -a')")
```

（3）反序列化一个 pickle 对象时。如果 pickle 反序列化用户在登录信息时没有进行校验，则当对应的存储介质（memcache、redis）没有开启登录认证并且暴露在公网中时，攻击者可以轻易向其中注入代码。例如：

```
pickle.loads("cposix\nsystem\np0\n(S'uname -a'\np1\ntp2\nRp3\n.")
```

（4）执行一个 Python 文件。执行外部文件时十分容易发生代码注入。攻击者可以在外部文件中添加后门漏洞，在执行文件时对服务器进行攻击。例如：

```
execfile("testf.py")
```

2.代码注入攻击的防范措施

代码注入攻击的现象在许多编程语言中都十分常见。进行安全编码是每个程序开发人员都必须注意的。一般常用以下几种方法防范代码注入攻击：

（1）严格控制输入，过滤所有危险模块，遇到非法字符时直接返回。

（2）使用 ast.literal_eval()函数代替 eval()函数，在项目中禁止使用 eval()函数，可在 git hook 中添加代码检查。

（3）安全使用 pickle。

6.2.2 密码存储方式

在各种应用中，用户名和密码是用户身份认证的关键，因此密码安全保护的重要性不言而喻。一是因为密码作为保护服务器和用户敏感数据的钥匙，一旦被破解，系统将完全暴露在攻击者面前。二是由于大多数用户会在不同应用中使用近似甚至完全相同的密码，因此密码本身就是非常敏感的数据，一旦一个应用的密码被破解，攻击者可能使用此密码去尝试破解用户在其他网站的账号。

1.传统密码存储方式

在系统中存储用户密码有以下几种常见方法：

（1）系统完全不接触密码，将用户的身份认证转交受信任的第三方来处理。例如 OpenID 这样的解决方案，系统向受信任的第三方求证用户身份的合法性，用户通过密码向第三方证明自己的身份，这样系统就不需再负责保证密码的安全。对用户来说，他们也不需要为每个应用都注册账号，因为同一个 OpenID 就可以登录所有支持 OpenID 的系统。然而，用户资源在现在的网络世界十分有价值，大多数网站都希望自己掌握用户资源。

(2) 密码明文直接存储在系统中。既然应用希望掌握用户资源,则他们需要自己负责用户的认证。其中,安全系数最低,但是在现在又十分常见的一种方法是将密码明文存储。采用这种方法时,管理员能查看所有用户的密码明文,一旦管理员权限被盗取,后果将十分严重。

(3) 密码明文经过转换后再存储。将密码经过一些数学算法转换加密后再存储。这种方法与直接存储密码明文没有本质区别,任何知道或破解出转换方法的人都可以通过逆转换得到密码明文。

(4) 密码经过对称加密后再存储。采用此方法时,密码明文的安全性等同于加密密钥本身的安全性。由于对称加密的密钥需要同时用于加密与解密,所以一般它会直接出现在加密代码中,这样攻击者也可轻易破解密钥,进而解密得出密码原文。

(5) 密码经过非对称加密后再存储。采用此方法时,密码的安全性等同于私钥的安全性。网站将密码明文经过公钥加密后存储。如果攻击者想要还原密码明文,就必须破解出相应的私钥才行,因此,只要保证私钥的安全,就可以保证密码明文的安全。密码的私钥可以交由受信任的人或机构来掌管,正常的网站身份验证只需要用公钥就可以。

以上 5 种方法的共同特点是可以从存储的密码形式还原到密码明文。然而,只要从网站系统能知道密码明文,那么攻入这个网站的攻击者也可以对网站用户密码进行还原,所以密码最好是以不可还原明文的方式来保存。

2. Python 中的密码存储方式

在 Python 中,通常利用哈希算法的单向性来保证明文以不可还原的有损方式进行存储。这类方法的各种具体操作方式按安全性由低到高依次如下:

(1) 使用网站独创的哈希算法对密码进行哈希运算,存储运算后的值。然而哈希算法十分复杂,独创哈希算法对密码学理论知识要求很高,一般独创的哈希算法并没有已公开的经过时间检验的哈希算法质量高,因此并不十分安全。

(2) 使用 MD5 或 SHA-1 算法。这是目前比较常见的加密方法。但由于近些年 MD5 和 SHA-1 已被破解,攻击者虽不能还原密码明文,但可以轻易找到能生成相同哈希值的替代明文。并且这两个算法速度较快,暴力破解相对省时。

(3) 使用更安全的 SHA-256 等成熟算法。这种算法更加复杂,且增加了暴力破解的难度。但如果用户密码过于简单,攻击者通过彩虹字典将很快就能得到密码原文。

(4) 加入随机盐(salt)的哈希算法。将密码原文(或经过哈希运算后的值)和随机生成的盐字符串混淆,然后再进行哈希运算,最后把哈希值和盐值一起存储。验证密码的时候只需要用盐值与密码原文做一次相同的运算,比较运算结果和存储的哈希值是否相同即可。这样,即使是简单密码,在经过加盐混淆后产生的也是并不常见的字符串,根本不会出现在彩虹字典中。盐字符串越长,暴力破解的难度越大。

6.2.3 异常处理机制

异常处理在许多编程语言中都是值得关注的一个话题。在操作系统提供的调用过程中,程序员往往通过事先约定一个返回的错误码来判断程序是否有错以及出错的原因。

例如,打开文件的函数 open()在成功时返回文件描述符(一个整数),在出错时返回-1。然而用错误码来表示是否出错在现在的程序运行过程中十分不便,因为函数本身应该返回的正常结果经常和错误码混在一起,造成调用人员需要额外使用大量代码来判断程序是否出错。所以大多数高级语言通常都内置了一套错误处理机制来帮助开发人员检查程序是否有异常,Python 也不例外。良好的异常处理机制可以让程序抵抗攻击的能力更强,增强程序的容错性。清晰的错误信息可以帮助程序员快速修复问题。

1. Python 的异常处理机制

通常,当 Python 解释器检测到错误,无法正常执行程序时,就会触发异常。如果异常被触发且没被处理,程序就会在当前异常处终止,后面的代码将不会运行。因此,当异常发生时,程序员需要编写特定的代码对异常进行捕捉和处理,如果捕捉成功,系统将进入另外一个分支,执行处理此异常的代码,使程序不会崩溃,这就是异常处理。

在 Python 中,常使用 try-except-finally 错误处理机制来进行异常处理。当程序员认为程序中的某段代码可能会出错时,可以使用 try 语句来运行这段代码,当开始一个 try 语句后,Python 就在当前程序的上下文中作标记,这样,当异常出现时就可以回到这里。如果 try 后的语句执行时发生异常,则后续代码不会继续执行,Python 直接跳回到 try 并执行第一个匹配该异常的 except 语句,让 except 语句捕获异常信息并进行处理,在执行完 except 语句后执行 finally 语句(可以没有 finally 语句),至此程序执行完毕。如果在 try 后的语句里发生了异常,却没有与之匹配的 except 子句,异常将被递交到上一层的 try,或者返回程序的最上层(这样将结束程序,并打印默认的出错信息);如果没有异常发生,则 except 语句不会被执行,程序将继续执行 finally 语句,直至结束。

Python 还可以使用内置的 logging 模块来记录错误,从而进行异常处理。系统可以把捕获的错误信息打印出来,分析错误原因,同时程序可以继续执行下去,并正常退出。通过配置,logging 还可以把错误记录到日志文件里,方便程序员事后进行进一步排查。

当发生了不同类型的错误时,应该由不同的 except 语句处理,主要有以下两种形式:

(1) 使用 except 处理所有异常类型。格式如下:

```
try:
    正常的操作
except:
    发生异常,执行这块代码
else:
    如果没有发生异常,执行这块代码
```

这种 try-except 语句可以捕获所有发生的异常。但这并不是一个很好的方式,因为它可以捕获所有的异常,所以程序员无法通过该程序识别出具体的异常类型。

(2) 使用 except 按异常类型分别处理。

可以使用相同的 except 语句分别处理多个异常信息。格式如下:

```
try:
    正常的操作
```

```
except(Exception1,Exception2,…,ExceptionN):
    发生以上多个异常中的一个,执行这块代码
else:
    如果没有发生异常,执行这块代码
```

2 异常处理中的常见错误

异常处理中的常见错误如下：

- 代码运行前的语法或者逻辑错误(用户输入不完整或输入非法)。
- AttributeError：试图访问一个对象没有的属性。例如，要访问 foo. x，但 foo 没有属性 x。
- IOError：输入输出异常，无法打开文件。
- ImportError：无法引入模块或包，通常是路径问题或名称错误。
- IndentationError：代码没有正确对齐。
- IndexError：下标越界。例如，x 只有 3 个元素，却试图访问 x[5]。
- KeyError：试图访问字典里不存在的键。
- KeyboardInterrupt：键盘中断，Ctrl+C 键被按下。
- NameError：尝试访问一个没有声明的变量。
- SyntaxError：语法错误，代码不能编译。
- TypeError：传入对象类型与要求的不符合。
- UnboundLocalError：试图访问一个还未被设置的局部变量，一般是因为程序内另有一个同名的全局变量，导致系统以为程序正在访问它。
- ValueError：传入一个调用者不期望的值(即使值的类型是正确的)。

6.2.4　文件上传漏洞的防御

文件上传漏洞是指用户在上传了一个可执行的脚本文件后，通过此脚本文件获得了执行服务器端命令的功能。这种攻击方式是最直接、有效的，对漏洞攻击了解不多的人也可以实施这种攻击。文件上传本身是没有问题的，但如果服务器处理和解析文件的模式不够安全，对恶意的可执行文件没有进行安全检测，服务器端就执行此文件，则会导致严重的安全问题。

1. 文件上传漏洞的原理

如果上传的文件是 Web 脚本，则服务器的 Web 容器将解释并执行上传的 Web 脚本，导致恶意代码被执行，这就是常见的 Web shell 问题。如果上传的文件是 flash 的策略文件，攻击者可以通过控制 flash 在该域下的行为来进行其他攻击。如果上传的文件是病毒或木马文件，攻击者可以通过服务器诱使其他用户下载并执行文件，以传播病毒。如果上传的文件是钓鱼图片或者包含了脚本文件的图片，则在某些版本的浏览器中，这些脚本文件会被执行，从而被用来实施钓鱼攻击。

要完成文件上传攻击，需要满足以下几个条件：

(1) 上传的文件需要能够被 Web 容器解释执行，因此文件上传的目录应该是 Web

容器所覆盖的路径。

(2) Web 服务器能够访问这个文件。如果文件上传后用户无法通过 Web 访问这个文件，那么攻击者即使上传了这个文件，也无法对用户进行攻击。

(3) 用户上传的文件没有被安全监测格式化或者被图片压缩等处理改变了内容。如果上传文件的内容被改变，则文件可能会解析失败，也就无法进行攻击了。

2. 文件上传攻击的防范措施

为了更好地防范文件上传攻击，可考虑以下几种措施：

(1) 将文件上传目录设置为不可执行。对于 Linux 而言，撤销其目录的可执行权限。许多大型网站的上传应用都会放置在独立的存储设备上，作为静态文件处理，这样一是方便使用缓存加速功能降低能耗，二是杜绝了脚本执行的可能性。

(2) 对上传文件类型进行检查。可以选择白名单，结合 MIME Type、后缀检查等方式对文件类型进行判断；对于图片文件的处理可以使用压缩函数或 resize() 函数在处理图片的同时破坏其包含的 HTML 代码。

(3) 使用随机数改写文件名和文件路径。这样一是使文件在上传后无法访问，二是 shell. php. rar. rar 和 crossdomain. xml 等文件都将因为被重命名而无法实施攻击。

(4) 单独设置文件服务器的域名。由于浏览器的同源策略，这种方法会使一系列客户端攻击失效，例如上传 crossdomain. xml、上传包含 JavaScript 脚本的 XSS 攻击等问题。

6.3 Python 安全编码规范

在开发过程时，程序设计人员通常更多地关注从需求到代码的设计实现以及开发的速度和质量，而最重要的安全性则不易被量化和关注。虽然大部分普通用户只会关注使用是否合理，但仍有一小部分用户会关注如何发现和利用系统的漏洞，甚至对系统加以破坏。因此，提高开发人员的安全编码意识以及进行安全编码十分重要。

本节基于 Python 主要发行版本的标准库，介绍 Python 安全编码规范，以帮助开发人员提升代码的可读性，保持项目中编码的风格一致性和安全性。需要注意的是，不要为了遵守编码规范而破坏代码的兼容性。

6.3.1 代码布局

1. 源文件编码

代码在 Python 3 中以 UTF-8 格式编码，在 Python 2 中采用 ASCII 编码。使用 ASCII 编码或 UTF-8 编码的文件应该有编码声明。

Python 3 和更高版本的标准库规定：Python 标准库中的所有标识符必须使用 ASCII 标识符，并在一般情况下使用英语单词（在许多情况下，缩写和技术术语是非英语的）。此外，字符串和注释也必须使用 ASCII 标识符。当测试非 ASCII 特征的测试用例以及编写作者的名字时例外。作者的名字如果不使用拉丁字母拼写，必须提供拉丁字母

的音译写法。

2. 导入

导入语句总是位于文件的开头，在模块注释和文档说明之后，在模块的全局变量与常量之前。导入通常按照以下顺序分组：标准库导入，相关第三方库导入，本地应用/库特定导入。通常应分开导入。例如，以下是推荐的写法：

```
import os
import sys
```

以下是不推荐的写法：

```
import sys, os
```

推荐使用绝对路径导入。如果导入系统时的路径没有正确地配置，使用绝对路径将会更清晰并且能更准确地提供错误信息。指定相对导入路径也是一个可接受的替代方案，特别是在绝对路径特别冗长的时候。

当从一个包含类的模块中导入类时，通常这样描述：

```
from myclass import MyClass
from foo.bar.yourclass import YourClass
```

如果上面的写法导致名字的冲突，则可以写为

```
import myclass
import foo.bar.yourclass
```

然后在引用时使用 myclass. MyClass 和 foo. bar. yourclass. YourClass 的形式。

3. 字符串引号

在 Python 中，使用单引号和双引号的字符串是相同的，选择一种引号并坚持使用下去即可。当一个字符串中包含单引号或者双引号字符的时候，使用和最外层不同的引号来避免使用反斜线进行转义，从而提高可读性。对于三引号字符串，总是使用双引号字符来与 PEP 257 中的文档字符串约定保持一致。

6.3.2　注释语句

注释语句在句尾结束的时候应该加两个空格。若注释很短，结尾的句号可以省略。如果注释是一个短语或句子，它的第一个单词的首字母应该大写，除非它是以小写字母开头的标识符（标识符的大小写不应改变）。

用英文书写时，应遵循规范的英文书写风格。非英语国家的 Python 程序员也应使用英文写注释。当代码被更改时，注意更新对应的注释。

1. 块注释

块注释一般是由完整句子组成的一个或多个段落，并且在每句话的结束处有个句号。

块注释通常缩进到与其需要解释的代码相同的级别。在块注释的每一行开头，使用一个＃和一个空格(除非块注释内部缩进文本)；在块注释内部的段落间，使用一个＃和空行分隔。

2. 行内注释

行内注释是与代码语句同行的注释。行内注释和代码之间至少要用两个空格分隔，注释由＃和一个空格开始。

应该有节制地使用行内注释。事实上，如果代码意义很明显，行内注释就是不必要的，因为它反而会分散注意力。例如，下面的注释就是不必要的：

```
x = x + 1                 # Increment x
```

但有时行内注释很有用，例如：

```
x = x + 1                 # Compensate for border
```

3. 文档说明

要为所有的公共模块、函数、类以及方法编写文档说明。非公共的方法不需要文档说明，但是需要一个描述此方法具体作用的注释，此注释应该在 def 行后。

对于单行的文档说明，结尾的三引号应该和文档说明的文字在同一行。对于多行的文档说明，结尾的三引号应该单独占一行。

符合规范的文档说明如下：

```
"""Return a foobang

Optional plotz says to frobnicate the bizbaz first.
"""
```

6.3.3 命名规范

Python 库没有统一的命名规范，但目前有一些推荐的命名标准，新的模块和包(包括第三方框架)应该采用这套标准；但对于已有的库，当其已采用了不同的命名风格时，推荐保持内部一致性。

1. 最重要的原则

对公众可见的命名应当优先考虑反映其用途，如 API 中的命名。

2. 命名约定

1）异常名

因为异常一般都是类，所以类的命名方法在这里也适用。但需要在异常名后面加上 Error 后缀(如果该异常确实是一个错误)。

2）全局变量名

全局变量应只在模块内部使用，它采用和函数一样的命名规则。通过 from M import * 导入的模块应该使用 all 机制来防止内部的接口对外暴露，或者使用在全局变量前加下画线的方式来表明这些全局变量是模块内非公有的。

3）函数和方法参数

始终将 self 作为实例方法的第一个参数。

始终将 cls 作为类静态方法的第一个参数。

如果函数的参数名和已有的关键字冲突，在参数名的最后加单下画线比缩写或随意拼写更好。因此 class_比 clss 更好。

4）方法名和实例变量

方法名和实例变量遵循这样的命名规则：使用下画线分隔小写单词以提高可读性；在非公有方法和实例变量前使用单下画线。Python 通过类名对这些命名进行转换。例如，类 Foo 有一个名为__a 的成员变量，它将无法通过 Foo.__a 访问。一般通过前缀双下画线以避免类中的属性名与子类名冲突。

5）继承的设计

始终要考虑到一个类的方法和实例变量（统称为属性）应该是公有的还是非公有的。如果存在疑问，那就选非公有，因为将一个非公有变量转为公有比反过来更容易。

另一种属性是子类 API 的一部分（在其他语言中通常被称为 protected，即受保护的）。有些类是专为继承设计的，用来扩展或者修改类的一部分行为。当设计这样的类时，要谨慎决定哪些属性是公开的，哪些是作为子类的 API，哪些只能在基类中使用。

以下是 Python 中关于继承的规范：

（1）公共属性不应该有前缀下画线。

（2）如果公共属性名和关键字冲突，应在属性名之后增加一个下画线。这比缩写和随意拼写好很多（当然，在作为参数或者变量时，cls 是表示类的最好的选择，特别是作为类方法的第一个参数时）。

（3）对于单一的公有属性数据，最好直接对外暴露它的变量名，而不是通过存取器（accessor）/突变（mutator）方法。请记住，如果一个简单的属性需要扩展为一个功能行为，Python 为这种将来会出现的扩展提供了一个简单的途径。在这种情况下，使用属性去隐藏属性数据访问背后的逻辑。

（4）如果允许一个类被继承，并且这个类里有不希望子类使用的属性，就要考虑使用前缀双下画线并且没有后缀下画线的命名方式。这会调用 Python 的命名转换算法，将类名加入属性名里。这样做有助于避免在子类中不小心包含了相同的属性名而产生的冲突。

3. 公共和内部的接口

在 Python 中，公共和内部的接口应遵守以下规范：

（1）向后兼容一般只适用于公共接口，因此，用户需要清晰地区分公共接口和内部接口。文档化的接口被认为是公共的，除非文档明确声明它们是临时接口或内部接口。所

有未文档化的接口都应该是内部接口。

(2) 为了更好地支持内省(introspection),模块应该使用__all__属性显式地在它们的公共 API 中声明名称。

(3) 将__all__设置为空列表,表示模块没有公共 API。

(4) 即使通过__all__进行了设置,内部接口(包、模块、类、方法、属性或其他名字)依然需要单下画线前缀。

(5) 如果一个命名空间(包、模块、类)被认为是内部的,那么包含它的接口也应该被认为是内部的。

6.3.4 函数安全

在使用函数时,应注意可能产生的安全问题。

1. ctypes

ctypes 是 Python 的一个外部库,它提供了和 C 语言兼容的数据类型。由于 ctypes 对内存大小没有限制,也不对溢出进行检查,在 32 位和 64 位操作系统上都可能造成溢出,因此,需对数据的有效性和溢出进行检查。

2. os

在下面的代码中,os. utime(path, times)的功能是设置对应文件的访问(access)和修改(modified)时间,时间以(atime, mtime)元组的形式传入。

```
import os
TESTFILE = 'temp.bin'
validtime = 2* * 55
os.utime(TESTFILE,(- 2147483648, validtime))
stinfo = os.stat(TESTFILE)
print(stinfo)
invalidtime = 2* * 63
os.utime(TESTFILE,(- 2147483648, invalidtime))
stinfo = os.stat(TESTFILE)
print(stinfo)
```

上面的代码中将 modified time 设置得过大,会产生报错。Modules 通常不对无效输入进行检验或者测试。例如,对于 64 位操作系统,最大数可以达到 $2^{63}-1$,但是在不同的情况下使用数值会造成不同的错误,任何超出有效边界的数值都会造成溢出,所以要对输入的数据进行检验。

3. len()函数

len()函数不对对象的长度进行检查,也不使用 python int objects。当对象可能包含一个“. length”属性的时候,就有可能造成溢出错误。这个问题的解决办法是使用 python int objects。

6.3.5　编程建议

在使用 Python 编程时,应注意以下几点:

(1) 在编写代码时需要注意不能影响程序在其他 Python(例如 PyPy、Jython、IronPython、Cython、Psyco 等)中的应用。例如,不要依赖于在 CPython 中高效的内置字符连接语句 a+=b 或者 a=a+b,因为这些语句在 Jython 中运行得很慢。在性能要求比较高的库中,应该改用''.join() 语句,这将保证程序在不同 Python 中运行时,字符串的连接时间仅取决于字符串的长度。

(2) 与 None 这样的单例对象进行比较时,应该始终用 is 或者 is not,不要用等号运算符。写 if x 的时候,请注意需要表达的意思是否是 if x is not None。例如,当测试一个默认值为 None 的变量是否被设置为其他值时,这里的其他值应该在上下文中能成为布尔型的 false。应该使用 is not 运算符,而不是 not … is。虽然这两种表达式在功能上完全相同,但前者更易于阅读,所以应优先考虑。例如,以下是符合规范的代码:

```
if foo is not None:
```

以下是不推荐的代码:

```
if not foo is None:
```

(3) 当使用比较实现排序操作的时候,最好使用下列 6 个操作符:__eq__、__ne__、__lt__、__gt__、__le__、__ge__,而不是依靠其他的比较方法,以免在其他版本中出现冲突。

(4) 始终使用 def 表达式,而不是通过赋值语句将 lambda 表达式绑定到一个变量上。例如,以下是符合规范的代码:

```
def f(x): return 2*x
```

而以下是不推荐的代码:

```
f = lambda x: 2*x
```

第一个形式意味着生成的函数对象的名称是 f 而不是泛型＜lambda＞,而 f 在回溯和字符串显示的时候更有用。而赋值语句的使用消除了 lambda 表达式优于 def 表达式的唯一优势(即 lambda 表达式可以内嵌到更大的表达式中)。

(5) 设计异常的等级时,要基于捕捉异常代码的需要,而不是异常抛出的位置。程序员应该以编程的方式回答"出了什么问题?",而不是只是确认"出现了问题"。当捕获了异常时,应尽可能写上具体的异常名,而不是只用一个 except 代码块。如果只有一个 except 代码块,将会捕获 SystemExit 和 KeyboardInterrupt 异常,这样就很难通过按 Ctrl+C 键中断程序,而且会掩盖其他问题。如果想捕获所有指示程序出错的异常,应该

使用 except Exception:的形式(只有 except:时等价于 except BaseException:)。

以下两种情况不应该只使用 except 代码块:

① 如果异常处理的代码会打印或者记录日志。

② 如果代码需要做清理工作,使用 raise-try-finally 能很好地处理这种情况,并且能让异常继续上浮。

(6) 当需要给捕捉的异常绑定一个名字时,为了避免和原来基于逗号分隔的语法出现歧义,推荐使用在 Python 2.6 中加入的显式命名绑定语法。捕捉操作系统的错误时,推荐使用 Python 3.3 中 errno 内定数值指定的异常等级。另外,对于所有的 try-except 语句块,在 try 语句中只填充必要的代码能避免掩盖代码缺陷。当代码片段局部使用了某个资源的时候,使用 with 或 try-finally 表达式来确保这个资源使用完后被清理干净。

(7) 无论何时要获取和释放资源,都应该通过单独的函数或方法调用上下文管理器。

(8) 返回语句应保持一致。函数中的返回语句或者都应该返回一个值,或者都不返回值。如果一个返回语句需要返回一个值,那么在没有值可以返回的情况下,需要用 return None 显式指明,并且在函数的最后显式指定一条返回语句。

(9) 使用字符串方法代替字符串模块。因为字符串方法总是更快,并且和 Unicode 字符串使用相同的 API。使用''.startswith()和''.endswith()代替通过字符串切割的方法去检查前缀和后缀。因为 startswith()和 endswith()出错概率更小。对象类型的比较应该用 isinstance()而不是直接比较 type 值。对于序列(包括 strings、lists 和 tuples)来说,可以使用空序列为 false 的情况。

(10) 书写字符串时,在转行处不要用空格分隔单词,这样的空格无法看到,有些编辑器会自动去掉这些空格。不要用==将一个值与 true 或者 false 比较。

第7章 软件安全测试

随着信息化建设的加快,软件系统已经覆盖到社会生产、生活的方方面面,但随之产生的软件安全问题也日趋严重。因此,软件安全测试作为验证软件是否满足安全要求的一个环节,具有至关重要的作用。本章首先介绍软件安全测试的基本概念,其次阐述软件安全测试的流程和常用测试方法。本章的目标是帮助读者了解软件安全测试的重要性,明确软件安全测试过程及作用。

7.1 安全测试概述

安全测试是鉴别信息系统数据保护和功能维护的过程。安全测试需要涵盖6个基本安全概念:保密性、完整性、权限(身份验证)、授权(权限分配)、可用性、不可抵赖性。随着软件系统的不断普及,软件安全已成为评判软件质量的一个重要指标,软件安全测试则成为验证产品是否满足安全要求的一个重要环节,并为管理层判断产品是否可以发布提供信息。换言之,软件安全测试是在产品发布之前验证系统是否满足安全需求,同时发现系统的安全漏洞,并最终把这些漏洞的数量降到最低的一系列过程。

软件安全测试的定义有狭义和广义之分。狭义的软件安全测试是系统测试中的软件安全性测试,指的是执行安全测试用例的过程;广义的软件安全测试是指所有关于安全性测试的活动,它贯穿于整个软件开发生命周期,并且在不同的开发阶段采取不同的安全测试策略和方法。本节将从测试对象、测试目标、测试角度、测试方法4个方面,对广义的软件安全测试进行详细阐述。

1. 测试对象

软件安全测试的对象不能简单地理解为代码,它还包括与软件系统相关的各种文档。同时软件安全测试的对象也不是一成不变的,在不同的开发阶段,有不同的测试对象。例如,在需求分析阶段,软件安全测试需要对需求文档中的安全需求进行评审;在设计阶段,软件安全测试需要对设计文档、攻击面分析、威胁建模等文档进行评估;在编码阶段,软件安全测试需要对源代码进行审核等。

2. 测试目标

软件安全测试的根本目标是保证被测软件在面临恶意攻击时仍能按照可接受的方式运行。具体来说,在软件开发生命周期的各个阶段进行软件安全测试,以期通过及时找到漏洞并进行修复来降低成本,避免在软件后期维护中进行成本高昂的补救,同时提升软件

的安全质量,为度量软件的安全性提供数据。在软件开发生命周期不同阶段的漏洞修复代价示例如图 7-1 所示。

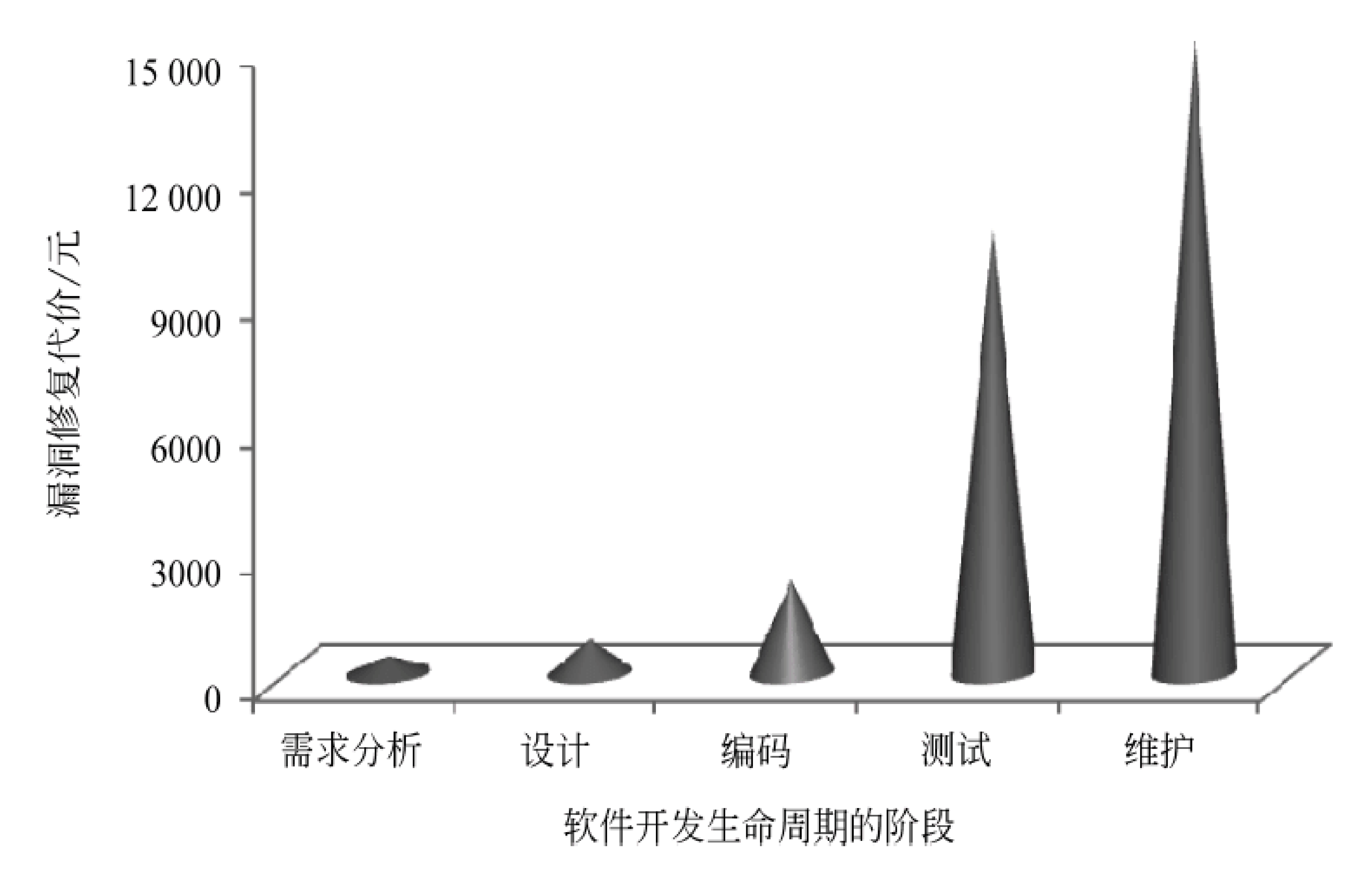


图 7-1　软件开发生命周期不同阶段的漏洞修复代价示例

3. 测试角度

软件安全测试角度与传统软件测试角度有本质上的不同。后者注重功能性测试,强调软件应该做什么,主要从最终用户的角度出发发现缺陷并进行修复,确保软件的功能符合用户的需求;前者注重安全性测试,强调软件不应该做什么,主要从攻击者的角度出发发现漏洞并进行修复,确保软件不被恶意攻击者破坏。一般情况下,普通用户仅会使用软件系统提供的基本功能,恶意攻击者则会通过各种途径寻找软件中的安全漏洞并进行攻击性操作。因此,安全测试工程师需要以恶意攻击者的思维对系统进行测试,例如,上传可执行文件至服务器并执行,或越权访问系统数据等。

软件安全测试与传统软件测试侧重点的不同,使得两者的测试用例也有所不同。传统软件测试根据功能需求和其他开发文档等,从用户角度设计测试用例,并且选取正向数据作为测试数据;软件安全测试根据安全需求、攻击模式归纳以及已公布漏洞等,从攻击者角度设计测试用例,并且多数选取反向数据作为测试数据。其中,正向数据指用户输入的正常数据,反向数据指攻击者构造的具有攻击性的非法数据。

在安全测试中,安全测试工程师应该模拟攻击者可能采取的所有攻击方式来探测系统,以期验证系统是否符合安全需求,这就意味着安全测试受限于已识别的漏洞和安全测试工程师的安全专业知识。安全测试在更多的时候测试的是反向的需求,例如,“攻击者应该不能篡改网页内容”“未授权用户不能访问数据”等;有时安全测试也会测试正向的需求,例如,“客户端与服务器端通信时数据必须加密”“三次登录不成功后账户将被锁定”等。

4. 测试方法

根据测试重点的不同,安全测试方法也有所不同,可分为黑盒测试、白盒测试和灰盒测试 3 种。

黑盒测试也称功能测试、数据驱动测试或基于规格说明书的测试，是一种从用户观点出发的测试，例如渗透测试。在安全测试中，把程序看作一个不能打开的黑盒子，在完全不考虑程序内部结构和处理过程的情况下，在程序接口进行测试，它只检查程序功能是否按照规格说明书的要求正常运行，程序是否能够适当地接收输入数据而产生相应的输出。黑盒测试着眼于程序外部结构，不考虑内部逻辑结构，因此它对测试者的技术要求不高，但是无法发现外部特性本身的设计问题。

白盒测试也称结构测试、逻辑驱动测试或基于代码的测试，是针对被测单元内部是如何进行工作的测试，例如源代码分析。在安全测试中，把程序看作一个透明的盒子，检查程序的内部逻辑结构，对所有的逻辑路径进行测试，分析程序的实现是否正确。白盒测试可以对代码进行详细审查，揭示隐藏在代码中的错误，但是它成本昂贵，并且无法检测代码中遗漏的路径和数据敏感性错误。

灰盒测试介于白盒测试和黑盒测试之间，多用于集成测试阶段，既关注输出对于输入的正确性，又关注程序内部逻辑结构。灰盒测试不如白盒测试详细、完整，但比黑盒测试更关注程序的内部逻辑结构，常常通过一些表征性的现象、事件、标志类判断内部的运行状态。

在实际测试过程中，通常同时使用黑盒测试和白盒测试进行互补测试，例如，通过黑盒测试找到可能存在的漏洞，然后进行代码审查，找到最终问题等。

7.2 安全测试流程

为更好地保证软件安全，需要制定一套标准和有效的流程来规范安全测试活动。同时，为提高安全测试的效率，可以将安全测试和 PDCA 循环进行结合。本节将详细阐述安全测试的具体流程、原则和内容以及安全测试与 PDCA 循环的有机结合。

7.2.1 安全测试具体流程

在实际测试过程中，软件安全测试流程一般包括以下 4 个阶段：计划阶段、设计阶段、执行阶段和总结阶段。

1. 计划阶段

计划阶段是安全测试流程的第一阶段。其主要工作是统筹规划所有安全测试活动，并根据测试计划目标制订安全测试计划。

测试计划目标在 IEEE 829—2008《软件测试文档标准》中已有明确的定义：规定测试活动的范围、方法、资源和进度；明确正在测试的项目、要测试的特性、要执行的测试任务、每个任务的负责人以及与计划相关的风险。因此，在计划阶段，需根据项目的整体计划、开发计划、安全计划和需求文档，明确需要测试的安全需求和安全测试环境，同时完成测试环境搭建活动的安排和跟踪，进而形成一份安全测试计划文档。

安全测试计划文档的内容主要包括基本简介、测试需求和风险、测试阶段和类型、资源、里程碑、可交付物以及风险 7 个部分。各部分具体内容如下：

(1) 基本简介。包括测试目的、测试背景、测试范围和测试依据。

(2) 测试需求和风险。需明确在基于风险的安全测试策略下的需求,并明确需求的优先级。

(3) 测试阶段和类型。需明确在单元测试、集成测试、系统测试以及验收测试阶段的测试类型。

(4) 资源。包括人力资源、系统资源和测试工具。

(5) 里程碑。需明确各时间段的工作内容和人员安排。

(6) 可交付物。需明确测试之后需交付的相关文档。

(7) 风险。需明确测试过程中可能遇到的风险,并对风险进行优先级划分,提出有效的应对措施。

需要注意的是,不能将安全测试计划简单地理解为一份文档,它更是一个过程,一个以交流软件测试小组意图、期望以及对将要执行任务的理解为目的的过程。只有明确了这一点,才能为后续阶段奠定坚实的基础。

2. 设计阶段

设计阶段是安全测试流程的第二阶段。该阶段需要针对不同测试类型设计不同的测试方案。一般情况下,根据测试类型的不同,可将测试方案分为两类。第一类是对功能的安全测试,可设计相应的测试用例。值得注意的是:在设计安全测试用例时与传统软件测试有所不同。在传统软件测试中,测试用例侧重功能实现与否;而在安全测试中,除了关注功能实现与否之外,还需关注安全功能实现的强度和安全功能实现的完备性,例如,测试日志记录功能实现的完备性。第二类是渗透测试、源代码分析等具体测试方法,此时则需要制订具体的测试方案。一份完整的测试用例主要包括用例描述、操作步骤、预期结果和执行步骤 4 个部分,必要时需增添测试目的、测试前置条件等项目。

3. 执行阶段

执行阶段是安全测试流程的第三阶段。在该阶段中,需要执行上一阶段设计的测试用例和测试方案,并记录执行的结果。执行结果可分为通过、失败、阻止和不支持 4 种,如表 7-1 所示。

表 7-1　执行结果及含义

执行结果	含　　义
通过(pass)	执行结果和预期结果一致
失败(fail)	执行结果和预期结果不一致,需判断是否存在漏洞,如果存在漏洞,需要在漏洞管理系统中提交
阻止(block)	由于某个原因导致当前用例无法执行
不支持(na)	当前测试环境无法执行当前用例

在执行安全测试过程中,如果执行结果与预期结果不一致,并且判断存在漏洞,则需要对漏洞进行完整记录。一份完整的漏洞报告需要包括标识符、漏洞描述、严重性和优先级 4 部分内容,具体含义如下:

(1) 标识符：识别一个漏洞的唯一编号。

(2) 漏洞描述：发现漏洞过程的详细描述。

(3) 严重性：漏洞可能对系统造成的影响。

(4) 优先级：漏洞修复的优先级别。

其中，判断漏洞优先级的典型方法是微软公司的 DREAD 模型。DREAD 是微软公司的计算机安全威胁分类系统中的一个计算模型。

DREAD 评估模型最初用于评估威胁，但在实际应用中远不止如此。每一项评估条目包含 1～3 个评分标准(其中 1 最低，3 最高)，根据计算后的结果可以安排安全威胁的解决工作优先级，即分数越高，优先级别越高。典型的威胁评价表如表 7-2 所示。

表 7-2　典型的 DREAD 模型威胁评价表

评价	高(3)	中(2)	低(1)
危害性	攻击者可以暗中破坏安全系统，获取完全信任的授权，以管理员的身份运行程序，上传内容	泄露敏感信息	泄露价值不高的信息
重现性	攻击每次可以重现，而且不需要时间间隔	攻击每次可以重现，但只在一个时间间隔和一个特定的竞争条件下才能进行	攻击很难重现，即使攻击者很了解安全漏洞
可利用性	编程新手在短时间内就可以进行这类攻击	熟练编程人员可以进行这类攻击，然后重复进行这些步骤	这类攻击需要经验丰富的人员才能进行，并且该人员应该对每次攻击都有深入的了解
受影响用户数	所有的用户，默认配置，主要客户	部分用户，非默认配置	极少的用户，特点不明确，影响匿名用户
可发现性	公开解释攻击的信息。可以在最常用功能中找到的缺陷	产品中较少使用部分的缺陷，只有少量的用户可能遇到。判断是否是恶意使用需要花费一些时间	缺陷不明显，用户不可能遭受潜在的损失

可以使用简单的高、中、低级别区分威胁的优先处理顺序，例如，将总分为 12～15 的威胁评价为高级威胁，将总分为 8～11 的威胁评价为中级威胁，将总分为 5～7 的威胁评价为低级威胁。如果某威胁被评价为高，说明其对软件系统造成的危险很大，需要尽快进行处理；如果某威胁被评价为中，也需要进行处理，但优先级别低于高级威胁；如果某威胁被评价为低，可以将其忽略，这取决于处理该威胁所需要的工作量和成本。

4. 总结阶段

总结阶段是安全测试流程的最后一个阶段。在该阶段中，需要报告整个安全测试活动的所有测试过程和测试结果，与预期结果进行比对，并对本次安全测试进行评估。

7.2.2　安全测试具体内容

安全测试用于验证产品是否符合安全需求，因此，根据软件系统的安全需求，软件安

全测试主要包括安全功能验证、安全策略验证、威胁缓解措施验证和系统实现安全验证 4 个方面的内容。

1. 安全功能验证

安全功能指为了系统能够安全运行而提供的必要的功能，例如，访问控制、加密算法和运行环境等。值得注意的是：区别于传统测试，在软件安全测试中，安全功能的实现仅仅是第一层次，除此之外还需要验证安全功能实现的强度和完备性。其中，安全功能实现验证指验证定义的安全功能是否能够有效实现；安全功能实现强度验证指在安全功能有效实现的基础上，验证安全功能的强度是否达到预期要求，例如，用户口令要求包含字母、数字和特殊字符三种类型；安全功能实现完备性验证指测试是否存在通过其他渠道降低安全实现的强度或者绕过安全实现的功能区域的情况。在实际测试过程中，可通过渗透测试工具发现 SQL 注入漏洞、XSS 漏洞等安全漏洞。渗透测试工具种类繁多，常见的有 Wireshark、X-Scan、Metasploit、Sqlmap、Burpsuite 等。虽然渗透测试工具非常多，通过信息收集，能够帮助测试人员高效地发现安全漏洞，但是对于一些系统功能上的漏洞却很难通过渗透测试工具判断，因此需要软件安全测试人员针对此类漏洞设计测试用例用于测试执行。

2. 安全策略验证

安全策略指在某个安全区域内(通常指属于某个组织的一系列处理和通信资源)用于所有与安全相关的活动的一套规则。在软件安全测试中，安全策略主要指针对整体的安全威胁和缓解措施规定的整体安全策略，例如，如何进行会话管理、日志管理等。同样，在安全策略验证过程中，测试人员需要完成两方面的测试工作：首先是安全策略的实现验证，分析规定的安全策略在各个测试层是否有效实现；其次是安全策略的实现完备性验证，测试是否有其他方式绕过安全策略，进而对软件系统造成威胁。

3. 威胁缓解措施验证

软件威胁建模作为一种应用风险评估技术，可以帮助系统设计师思考系统或应用程序可能面临的安全威胁。在实际应用中，它能有效帮助系统设计师开发出缓解潜在的漏洞威胁的策略，并帮助他们将有限的资源和注意力集中在系统中最需要进行安全测试的部分。威胁缓解措施则是针对威胁建模中发现的威胁而提出的解决方案。威胁缓解措施验证包括两个部分：缓解措施实现的验证和缓解措施实现的完备性验证。其中，缓解措施的实现验证测试缓解措施是否有效；缓解措施完备性验证测试是否有其他途径可以绕过缓解措施，使用此缓解措施是否会引起其他的安全问题。

4. 系统实现安全验证

系统实现安全验证用于测试整个系统的实现是否安全，是否存在严重的安全漏洞，是从整体上对系统进行的安全测试。软件安全测试的内容往往非常多，实际上，对系统中所有部分进行安全测试也许在时间和成本上是不可行的，因此需要对每一部分进行威胁评估。首先对威胁较大的部分进行安全测试，再对其他部分进行安全测试，并且可通过决策忽略一些部分，因为其威胁发生的概率很小，即使发生，带来的损失也很小。威胁评估过

程采用以下公式评估危险的大小：

危险＝威胁发生的概率×潜在的损失

该公式表明，特定威胁造成的危险等于威胁发生的概率乘以潜在的损失，这表明了如果攻击发生将会对系统造成的影响。通过威胁评估进行决策，实现软件安全测试活动的高效执行。

7.2.3 安全测试原则

软件的不安全问题也许是这个时代最为重要的技术挑战。安全问题是目前制约信息技术发展的关键。为更好地指导对软件系统安全性的验证，OWASP 在测试指南中总结了多项软件安全测试应遵循的基本原则，具体如下：

（1）安全是一个过程，不是某个产品。当开发者在面对安全扫描器或应用防火墙既不能提供各类攻击防御措施也无法辨别各类安全威胁的问题的时候，应该明白并不存在一步就能解决不安全软件问题的方法。安全不是一蹴而就的，它需要一个漫长的过程。

（2）有组织、有计划地进行软件安全测试。将安全融入软件开发生命周期的每一个阶段，制订详细的软件安全测试计划，并且严格按照计划进行。

（3）应及早进行软件安全测试，并进行频繁测试。在软件开发生命周期中尽早查出漏洞，可以使漏洞迅速地被修补并且花费较低的成本。

（4）明确项目的安全范围。对需要被保护的资料和资产应予以分类并分别确定处理方式，例如，将需要被保护的资料和资产分为保密（Confidential）、秘密（Secret）和绝密（Top Secret）3 个级别。

（5）测试人员应形成创造性思维。在正常模式下对应用程序的正常行为进行测试仅适用于用户按照开发人员所预期的规则来使用应用程序的情况，因此成功地测试一个应用程序的安全漏洞需要创造性思维。良好的安全测试需要测试人员拥有创新思想，从恶意攻击者的角度思考，从而发现安全漏洞。

（6）合理安排模块测试优先级。在有限的时间和资源条件下，进行完全测试，找出软件的所有缺陷是不可能的，应该通过威胁建模等方法优先测试高风险模块。

（7）适当情况下使用源代码。为更好地确保软件系统的安全，在适当情况下，应考虑将源代码移交给安全测试人员以帮助其进行测试工作。

（8）对测试结果进行文档记录。形成一份完整的测试报告，包括测试人员、测试时间、测试行为、测试结果等，明确指出软件系统存在的安全漏洞，以支持后续的漏洞排查行为。同时，值得注意的是，测试报告需明确指出安全漏洞的影响以及解决方案。

7.2.4 PDCA 循环

遵循软件安全测试流程 4 个阶段的要求及相关原则，能够有效地计划、组织、实施所有测试相关活动，但无法对测试流程进行改进，因此，可结合 PDCA 循环，根据上一阶段的评估结果实现对测试流程的改进。

PDCA 循环是基于过程方法制定的一种持续改进模型，又称戴明环。该模型最初由美国质量管理专家哈特休博士提出，后由美国质量管理专家戴明博士再度挖掘，并作为全

面质量管理的思想基础和方法依据加以推广，使 PDCA 模型得到广泛应用。PDCA 由英文单词 Plan、Do、Check 和 Act 的首字母组合而成，因此，PDCA 循环是将质量管理分为 4 个阶段，即计划（Plan）、执行（Do）、检查（Check）和处理（Act），各阶段具体工作如下：

（1）计划。评估当前状况，针对现状发现问题，分析产生问题的原因，建立必要的活动目标，制订有效的计划和对策。

（2）执行。按照预定的计划和标准，根据已知的内外部信息，设计具体的行动方案，进行布局，再根据设计方案和布局进行具体操作，努力实现预期目标。

（3）检查。评估行动方案的效果，对方案执行结果进行总结分析，并与目标值进行比较，分析是否达到预定目标。

（4）处理。根据检查结果，采取相应措施，对已被证明的有效措施进行标准化，将仍存在的问题转入下一个 PDCA 循环解决。

PDCA 模型不仅适用于质量管理工作，同样也适用于其他各项管理工作。该模型在应用时，计划、执行、检查和处理 4 个阶段并不是运行一次就完结，而是周而复始地运行。具体来说，按照 P→D→C→A 的顺序依次执行，一次完整的 PDCA 循环可以看作组织管理上的一个周期，每经过一次 PDCA 循环，组织管理体系就会得到一定程度的完善，同时进入下一个更高层次的管理周期。通过持续不断的 PDCA 循环，组织管理体系得到持续改进，管理水平随之不断提高。PDCA 模型应用过程如图 7-2 所示。

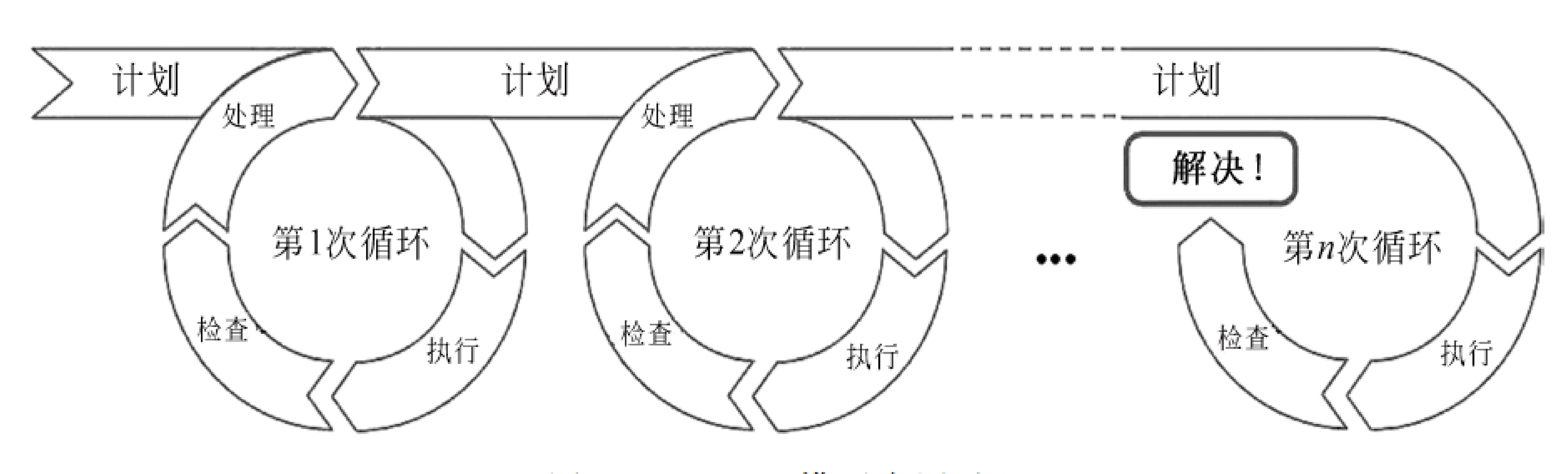


图 7-2 PDCA 模型应用过程

PDCA 循环作为一种广泛应用的管理工作方法，具有以下三大特点：

（1）依序进行，循环运转。PDCA 循环靠组织的力量来推动，按顺序完成每一阶段的工作，不断前进；与此同时，周而复始，不断循环。

（2）大环套小环，小环保大环，相互制约，相互补充。上一级循环是下一级循环的依据，下一级循环是上一级循环的组成部分，也是上一级循环的落实和具体化。也就是说，PDCA 循环通过各个小循环的不断运转，将组织的管理工作有机地结合起来。

（3）不断前进，不断提高。PDCA 循环每执行一次就上升一个台阶，犹如爬楼梯。每经过一次循环，一些问题就会得到解决，质量和管理水平也会上升一个高度，同时进行总结，提出新的目标，在新的基础上继续 PDCA 循环。如此周而复始，不断解决问题，质量和管理水平不断得到改进和提高。

在安全测试活动中，制订安全测试计划对应 PDCA 中的计划阶段；按照计划要求执行测试，包括设计阶段和执行测试阶段，对应 PDCA 中的执行阶段；总结报告，对各种文档和测试结果进行评估，对应 PDCA 中的检查阶段；根据上一阶段的评估结果进行流程

改进对应 PDCA 中的执行阶段。将安全测试活动与 PDCA 循环进行结合，有利于提高安全测试的质量和效率，更好地保证软件安全。

7.3　安全测试技术

在软件安全测试过程中需要用到不同的技术，合理运用安全测试技术，能够有效地提升测试效率和测试质量。常见的软件安全测试技术有人工审查、代码分析、模糊测试和渗透测试。本节对这 4 种测试技术进行详细介绍。

7.3.1　人工审查

人工审查是在安全测试过程中通过人工审核的方式对应用开发过程中的人、策略和进程，包括技术决策和开发模型设计，进行安全检测。人工审查通常采取文件分析法，包括分析安全策略、安全需求、威胁建模文档等，也可以用对设计师以及系统所有者进行访谈的方式进行。虽然人工审查和人员访谈这两个概念十分简单，但是它们的确是最强大、最有效的可用技术。通过询问别人一件事如何运行，为什么采用当前的运行模式，能够帮助测试者快速确定是否存在显而易见的安全问题。人工审查是在软件开发生命周期过程中对软件进行测试并确保其实现安全策略和安全需求的为数不多的途径之一。但值得注意的是：在进行人工审查的过程中，建议使用“信任但必须验证”的模式，因为并不是每个人告诉或展示给测试人员的每件事都是准确的。人工审查对测试人员对安全进程和安全策略的了解以及专业安全知识和技能有较高要求，并且测试人员需要有前期充分的准备，才能在审查过程中发现更多的安全缺陷。人工审查具有使用灵活、适用范围广泛等特点，这些特点使其在代码安全策略、安全要求、构架设计等文件的审核中也能发挥重要作用。

7.3.2　代码分析

许多严重的安全漏洞不能被任何其他形式的分析或测试技术检测到。几乎所有的安全专家一致认为，没有任何检测方法可以取代代码分析。代码分析有利于发现以下安全问题：并发的问题，有缺陷的业务逻辑，访问控制的问题，后门、木马等其他形式的恶意代码。同时，代码分析对于查找可能存在的执行问题，例如某个需要输入验证的地方不能有效执行或打开控制进程失败，是十分有效的。

根据分析条件的不同，代码分析技术可分为两类：静态代码分析技术和动态程序分析技术。静态代码分析技术是在不执行计算机程序的情况下对代码进行的分析，动态程序分析技术则是通过在真实或虚拟处理器上运行程序并利用计算机软件进行分析的方法。

1. 静态代码分析

静态代码分析主要用在某版本程序源代码的分析上，也可以用在对象代码的分析上。与此同时，静态代码分析使用自动化工具进行测试，因此，它可以使用比开发者更多的安全知识更频繁地进行测试。静态代码分析的流程如图 7-3 所示，可总结为 3 个阶段：输入

(源代码)、测试(静态分析)以及输出(确认潜在的缺陷)。其中,在输入阶段,通常将源代码作为输入对象;在测试阶段,已经获得软件将要执行的命令,能够准确定位有问题的代码,并且软件开发者能够在代码审查之前执行代码;在输出阶段,确认 XSS 漏洞、SQL 注入漏洞等缺陷以及与应用程序相关的配置错误和补丁错误,限定问题的范围,确认操作环境或运行时的漏洞。

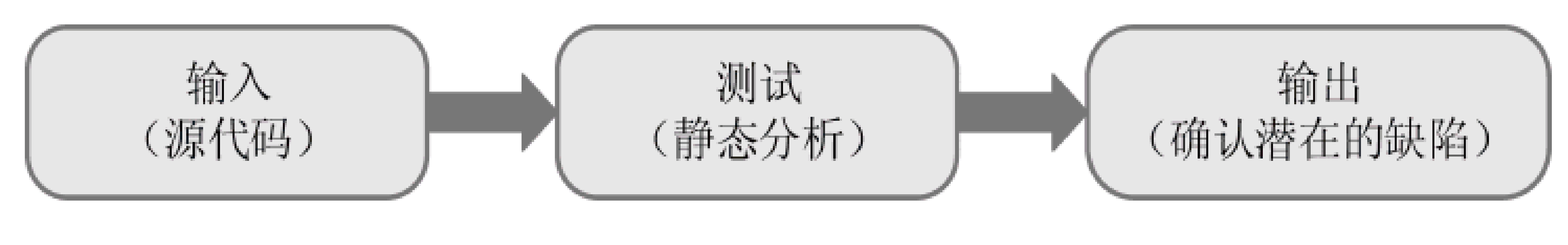


图 7-3　静态代码分析流程

静态代码分析技术大多以编译技术或程序验证技术为基础,目前常见的静态代码分析技术主要包括词法分析技术、抽象解释技术、程序模拟技术以及定理证明技术。

1) 词法分析技术

词法分析只对代码的文本或 Token 流与已经归纳好的缺陷模式进行相似性匹配,不深入分析代码的语义和代码上下文。词法分析技术检测效率较高,但是只能找到简单的缺陷,并且误报率较高。

2) 抽象解释技术

抽象解释技术的基本原理是将程序变量的值映射到更加简单的抽象域上并模拟程序的执行情况。因此,该技术的精度和性能取决于抽象域对真实程序值域的近似情况。值得注意的是:此类技术可用于证明一段代码没有错误,但不保证其报告的错误的真实性。

3) 程序模拟技术

程序模拟技术模拟程序执行,得到所有执行状态,分析结果较为精确,但性能提高难度大,主要用于查找逻辑复杂和触发条件苛刻的缺陷。程序模拟技术主要包括模型检查和符号执行两种技术。其中,模型检查技术将软件构造为状态机或者有向图等抽象模型,并使用模态/时序逻辑公式等形式化的表达式来描述安全属性,对模型进行遍历以验证这些属性是否满足;符号执行技术使用符号值表示程序变量值,并模拟程序的执行以查找满足漏洞检测规则的情况。

4) 定理证明技术

定理证明技术将程序错误的前提和程序本身描述成一组逻辑表达式,然后基于可满足性理论并利用约束求解器求得可能导致程序错误的执行路径。此类技术具有较高的灵活性,能够使用逻辑公式有效描述软件缺陷,并可根据分析性能和精度的不同要求调整约束条件,对于大型工业级软件的分析较为有效。

静态代码分析提供代码行级别的检测,确保开发组可以迅速修复安全漏洞。其成功的关键因素是静态分析工具的使用。不同静态代码分析工具的具体实现和工作原理不尽一致,但其核心都是在规则库的作用下识别漏洞,如图 7-4 所示。其中,规则库是影响静态分析效果的重要因素,也是静态分析工具选取时的主要考察内容之一,如果规则库内容全面且设置合理,则漏报、误报相对较少。与此同时,使用静态代码分析工具的另一个挑战是,当分析一个与闭源组件或外部系统进行交互的应用时,误报往往会写在报告中,由于没有源代码,不可能对外部系统中的数据流进行跟踪并确保数

据的整体性和安全性。

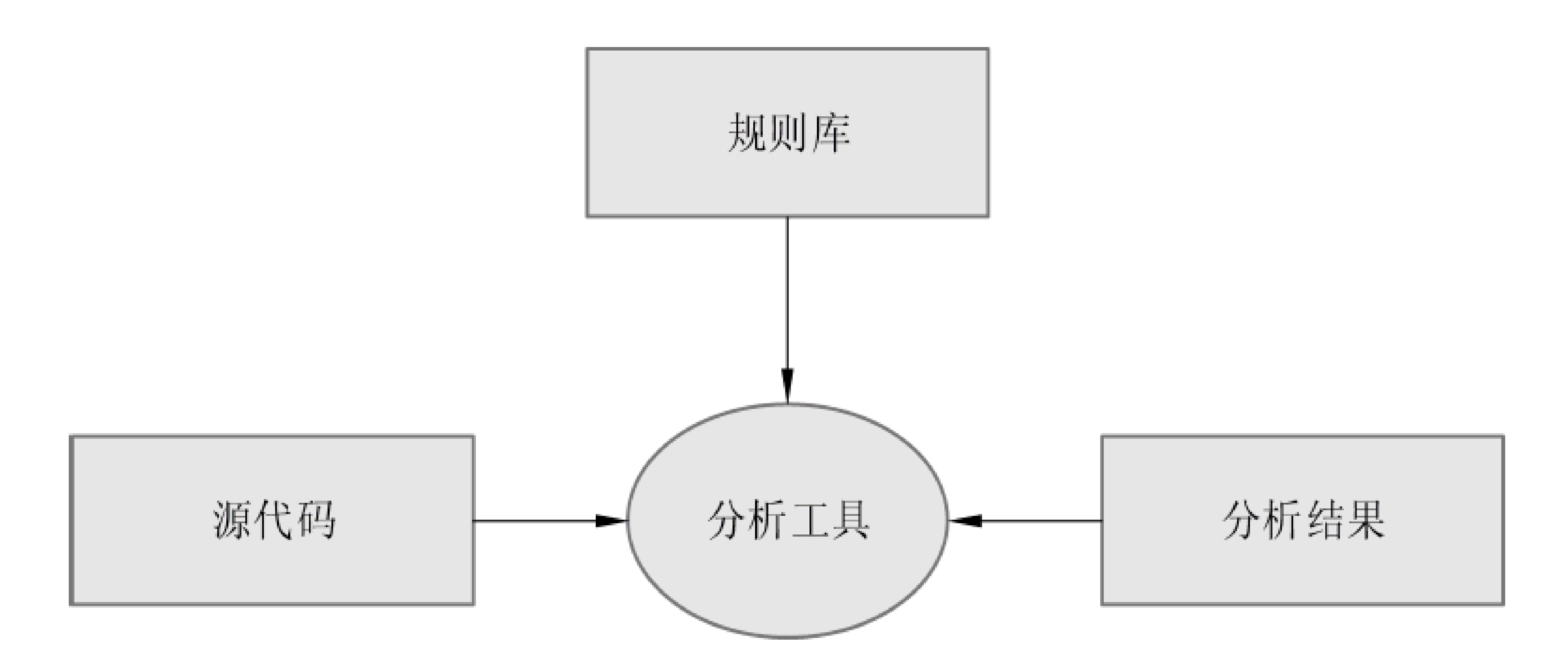


图 7-4　静态分析工具工作流程图

注入漏洞被 OWASP 列入十大安全漏洞列表中。注入漏洞发生的原因是不可信数据作为结构化的命令未经确认就直接用来查询。注入漏洞有不同的类别，如 SQL 注入漏洞、OS 注入漏洞和 LDAP 注入漏洞。如果直接用用户输入构建一个 SQL 查询，SQL 注入攻击就会成功。

以下是一个通过静态代码分析发现 SQL 注入漏洞的例子。用户检查他账户的细节。应用程序需要用他的用户 id 或标识符从后端数据库查询账户信息。应用程序可以通过一个 URL 链接参数绕过身份鉴别过程，例如：

```
http://example.com/advise/useraccount?account_id='1007'
```

在此例中，应用程序获得的用户 account_id 为 1007，并使用这个 id 获取数据库的信息。后端查询方式如下：

```
String insecureQuery="SELECT * FROM accounts WHERE
accountID='"+request.getParameter("account_id")+"'";
```

如果恶意用户将参数的值改为'or'1'='1，最终生成的 SQL 语句将会变成

```
SELECT * FROM accounts WHERE accountID=''or'1'='1';
```

'1'='1'永远为真，因此这个查询结果会产生所有账户信息。这显然不是开发者的意图，但是利用可信用户输入建立查询，恶意用户就可以执行任意数据库命令。

2 动态程序分析

动态程序分析技术在软件逆向工程领域也是一个非常热门的概念，它是与静态分析技术相对而言的，指通过观察程序的运行过程中的状态，如寄存器内容、函数执行结果、内存使用情况等，分析函数功能，明确代码逻辑，挖掘可能存在的漏洞。

逆向工程又称逆向技术，是一种产品设计技术再现过程，即对一项目标产品进行逆向分析及研究，从而推导并得出该产品的处理流程、组织结构、功能特性及技术规格等设计要素，以制作出功能相近，但又不完全一样的产品。逆向工程源于商业及军事领域中的硬件分析。其主要目的是在不能轻易获得必要的生产信息的情况下，直接从成品分析入手，推导出产品的设计原理。

随着计算机技术在各个领域的广泛应用，特别是软件开发技术的迅猛发展，基于某个软件，以反汇编阅读源码的方式去推断其数据结构、体系结构和程序设计信息成为软件逆向工程技术关注的主要对象。软件逆向技术的目的是研究和学习先进的技术，特别是当手里没有合适的文档资料，而又很需要实现某个软件的功能的时候。也正因为这样，很多软件供应商为了垄断技术，在软件安装之前，要求用户同意不对其进行逆向研究。

动态程序分析可分为 3 个阶段：输入（不需要源代码），测试（动态分析）以及输出（确认潜在的缺陷）。其中，在输入阶段，只需要运行应用程序，进行基础设施或操作配置，无须源代码；在测试阶段，不需要获得所有软件将要执行的实际指令，不能定位问题代码，需在操作环境下执行；在输出阶段，确认缓冲区溢出、SQL 注入漏洞、XSS 漏洞、内存泄漏、弱密码等缺陷，无法发现运行时环境或操作配置引入的安全漏洞，无法发现验证问题、业务逻辑缺陷或不安全的加密问题。动态程序分析流程如图 7-5 所示。

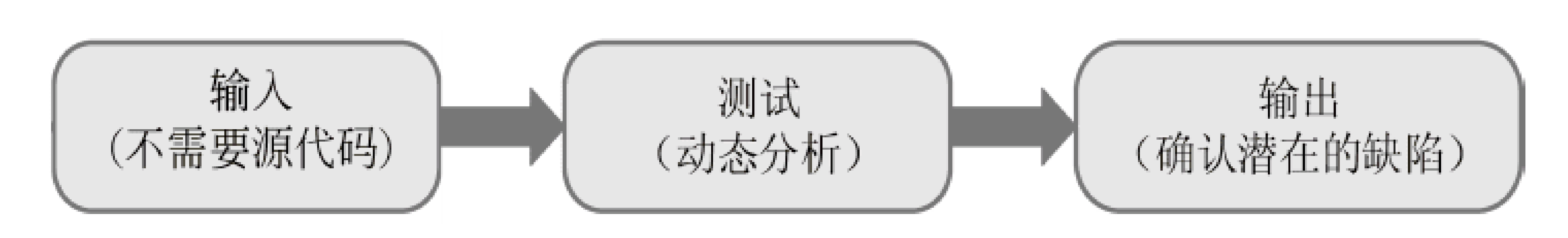


图 7-5　动态程序分析流程

在软件安全测试阶段，根据不同类型的测试方法和测试策略，可将动态分析分为 4 种测试类型：单元测试、集成测试、系统测试、验收测试。单元测试是对软件单元、模块或子程序的结构性或功能进行的测试。其中，结构性测试检测软件单元的逻辑，用来支持测试覆盖的需求——程序有多大程度被执行；功能性测试评估软件实现了多少软件需求。集成测试是在软件系统集成过程中进行的测试，其主要目的是检查软件单元之间的接口是否正确。系统测试是对已经集成好的软件系统进行彻底的测试，以验证软件系统的正确性和性能等满足其规约所指定的要求。验收测试是技术测试的最后一个阶段，也称为交付测试，是在软件产品完成了单元测试、集成测试和系统测试之后，产品发布之前所进行的软件测试活动，其目的是确保软件准备就绪，并且可以让最终用户将其用于执行软件的既定功能和任务。

动态测试形式使动态程序分析具有检测出静态代码分析检测不到的依赖关系的能力。例如，动态依赖使用反射依赖注入等，也可验证静态代码分析的结果。与此同时，可以使用自动化工具来提高扫描的灵活性，允许分析看不到源代码的应用程序，可在任何应用程序上执行。但动态程序分析也有其局限性，自动化工具只能达到扫描规则的水平，会存在误报和漏报，同时将安全漏洞回溯到代码中精确位置较为困难，需要花费较多时间来修复问题。在实际测试过程中，将静态代码分析与动态程序分析结合使用能有效提高测试效果。

7.3.3　模糊测试

模糊测试（fuzz testing）是一种软件安全测试技术。它在测试方法分类中可归于黑盒测试和灰盒测试领域。它通过监视非预期输入可能产生的异常结果来发现软件问题，其

核心思想是自动或半自动地生成随机数据，输入程序中，记录程序异常，如崩溃、断言失败等，从而对程序内部的缺陷做出判断，例如内存泄漏。模糊测试最早是由威斯康星大学 Barton Miller 教授于 1988 年提出的，他用一个纯随机的模糊器测试 UNIX 程序的健壮性。

1. 模糊测试的特点

模糊测试介于完全手工测试和完全自动化测试之间。在完全手工测试中，测试人员需模拟黑客行为，对系统执行恶意操作，从而发现漏洞，这对测试人员的要求较高，因而完全手工测试对测试人员能力的依赖性较强，成本较高，难以大规模实现；在完全自动化测试中，各个产品的需求、实现、功能等都有所不同，同一套测试用例和方法无法不加修改地用于不同的产品，因此在测试过程中，仍需测试人员介入，分析结果、判断漏洞等。在这种情况下，模糊测试得以发挥重要作用。

在模糊测试中，文件格式和网络协议是常见的测试对象。与此同时，任何程序输入都可以作为测试对象，常见的输入有环境变量、鼠标和键盘事件以及 API 调用序列。甚至一些通常不被考虑成输入的对象也可以用于测试，例如数据库中的数据或共享内存。模糊测试数据更多是无效或者半有效的。所谓半有效的数据是指：对于应用程序来说，测试用例的必要标识部分和大部分数据是有效的，这样待测程序就会认为这是一个有效的数据；但同时该数据的其他部分是无效的，这样，应用程序就有可能发生错误，出现冲突、锁住、消耗大量内存或者产生不可控制的程序错误，通过分析即能发现程序中存在的安全漏洞，从而在应用程序发布或者配置前予以矫正。

模糊测试可以有效地找出安全漏洞，是由于输入数据的随机性，因此不会被任何关于软件应该如何运作的偏见所束缚。但值得注意的是：模拟测试相当于对系统的行为进行一个随机采样分析，通过模糊测试说明软件可以处理某些异常情况，不会崩溃，但并不能说明该软件完全没有安全漏洞。模糊测试主要是对整体质量的一种保证，并不能替代全面的测试或者形式化方法。模糊测试可以提示程序的哪些部件需要特别注意，对于这些部件可以进一步使用静态代码分析等其他软件安全测试方法。

与传统漏洞挖掘方法相比，模糊测试技术有其独有的特点：

(1) 模糊测试是一种发掘漏洞的强制性方法。它不像白盒测试那样需要依靠软件开发文档，执行源代码分析；也不像黑盒测试那样关注测试目标对给定输入的输出是否符合预期。模糊测试只关注测试目标对测试用例是否作出了不当的反应，因此使用范围更加广泛。

(2) 模糊测试是动态实际执行的，不存在静态代码分析技术中存在的大量误报问题。同时，模糊测试的原理较为简单，没有大量的理论推导和公式计算，不存在符号执行技术中的路径状态爆炸问题。但需要注意的是：模糊测试用例多数是无效或者半有效的，在实际测试过程中，即使待测目标在处理这些测试用例(特别是半有效的用例)时产生了和正常数据一样的反应，对待测目标来说，这也是一种不正确的反应，是一种错误。

(3) 模糊测试中随机生成的测试用例只占很小的一部分，大部分测试用例需要靠人

的分析与经验进行有导向的生成，但这种导向是宏观的，具体的测试用例仍然由模糊器生成。因此，模糊测试技术自动化程度较高，不需要逆向工程中大量的人工参与。

2 模糊测试的 6个基本阶段

由于不同类型的待测目标差别很大，模糊测试针对不同的待测目标，在方法的选择上，根据不同的因素，会有很大的变化。选择模糊测试的方法取决于待测目标、测试者的经验以及待测数据的格式。但是，无论对何种目标进行模糊测试，也无论采用何种模糊测试方法，在模糊测试过程中都有 6 个基本阶段，分别是识别待测目标、识别预期输入、生成模糊测试用例、执行模糊测试用例、监视异常和故障以及检测可被利用的漏洞，如图 7-6 所示。

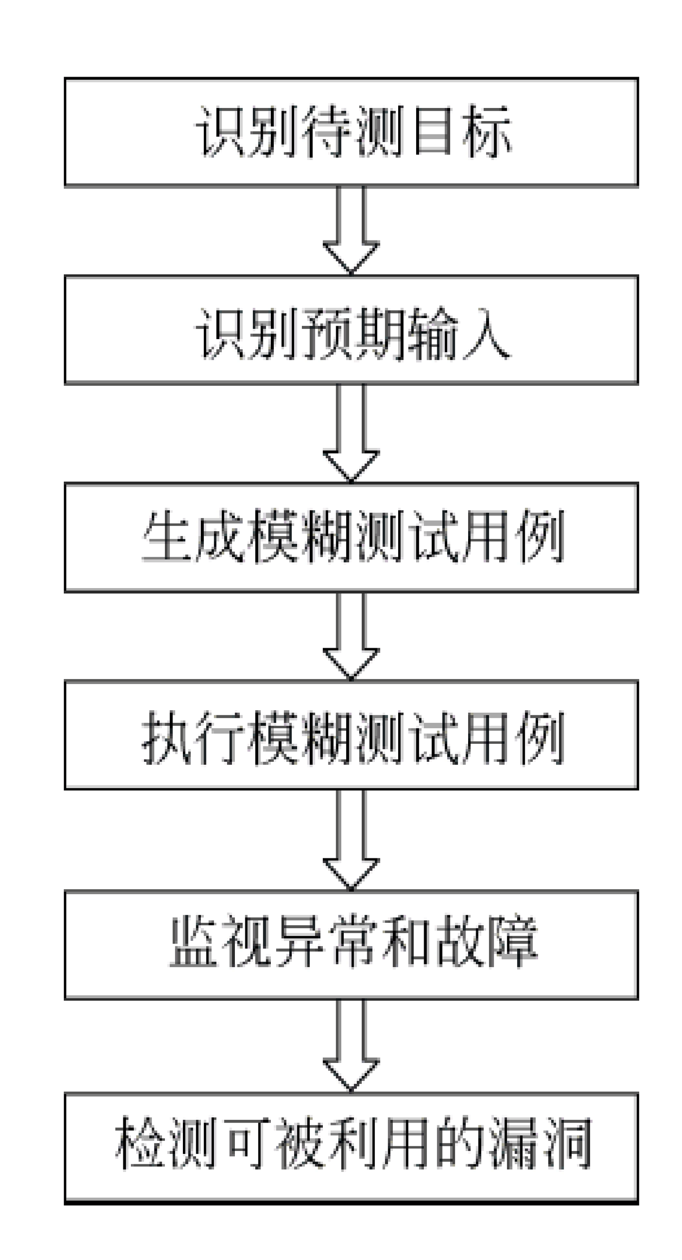


图 7-6　模糊测试的 6 个基本阶段

1）识别待测目标

模糊测试工作的第一步是确定待测目标，并根据目标的类型选取合适的模糊测试工具或者技术。同时，还可以借鉴基于风险模型的测试方法，根据风险分析的结果，对风险较高的应用或模块进行有针对性的模糊测试。当待测目标选定后，如果需要详细地确定某一个待测输入或待测文件格式，那么应该选取被较多的应用程序使用的输入或文件。因为它们被较多的程序访问和使用，所以它们出现漏洞的概率也相应较大。

2）识别预期输入

确定待测目标后，需要分析预期输入。模糊测试的测试用例都是基于对预期输入进行一定的模糊化，所以识别预期输入对模糊测试至关重要。当应用程序得到了用户的输入或其他来源的输入后，在处理这些输入的过程中没有预先分辨或拦截非法数据，就很容易出现安全漏洞。而执行模糊测试的过程就是不断地枚举输入向量的过程，针对不同的输入类型设计不同的模糊测试数据。常见的输入向量有消息头、文件名、环境变量、注册表键值等。任何从客户端发往目标应用程序的输入都应该作为输入向量。例如，对于一个 UDP 报文，不仅数据部分可作为模糊测试的输入，首部、文件名、模式、操作码、块编号、差错码、差错信息等也都是输入向量。

3）生成模糊测试用例

确定了输入向量后，就可以生成模糊测试用例。在实际测试中，需要根据待测目标及其数据格式选择合适的策略，使用不同的模糊器来生成模糊测试用例。由于数据量较大，这个阶段通常会采用自动化的方式完成。常用的模糊测试用例生成方法有预生成测试用例、随机生成输入、手工协议变异测试、变异或强制性测试以及自动协议生成测试。

（1）预生成测试用例的开发始于对一个专门规约的研究，其目的是理解所有被支持的数据结构和每种数据结构可接受的值范围。硬编码的数据包或数据文件随后被生成，以测试边界条件或迫使规约发生违例。预生成测试用例可用于检验目标系统实现规约的精确程度，并且在测试相同的协议的多个实现或文件格式时，用例能够被一致地重用。但是，创建这些测试用例需要事先完成大量的工作，同时，由于没有引入随机机制，一旦测试

用例表中的用例被用完，模糊测试只能结束。

(2) 随机生成输入只是简单地产生大量伪随机数据给待测目标，观察待测目标对这些数据的反应。该方法虽然效率较低，但适用于迅速地对待测目标进行宏观的安全性评估。

(3) 手工协议变异测试不需要自动化的模糊器，因为研究者本人就是模糊器。在加载了目标应用程序后，研究者仅仅通过输入不恰当的数据来试图让服务器崩溃或使其产生非预期的行为。该方法有利于测试人员在审核过程中发挥自身的历史经验和专业技术。

(4) 变异或强制性测试指模糊器从一个有效的协议或数据格式样本开始，持续不断地打乱数据包或文件中的每一个字节、字、双字或字符串。采用此方法，无须事先对被测件进行任何研究。但是，这种方法较为低效，许多 CPU 周期被用于数据生成，并且这些数据并不能立刻得到解释。然而，通过测试数据生成和发送过程的自动化，可以有效地提高测试效率。该测试方法的用例覆盖依赖于已知的、经过测试的、良好的数据包或文件，大部分协议规约或文件定义都比较复杂，即使对其进行表面的测试覆盖也需要相当数量的样本。强制性文件格式模糊器有 FileFuzz 和 notSPIKEfile，分别对应 Windows 和 Linux 系统。

(5) 自动协议生成测试是一种综合了前 4 种方法的、更高级的强制性测试方法。在这种测试中，需要进行前期的研究工作，首先要理解和解释协议规约或文件定义，创建一个描述协议规约如何工作的文法，然后由模糊器动态分析这些文法，生成模糊测试数据，向被测目标发送模糊测试后产生的包或文件。这种方法的成功依赖于测试者的能力，测试者需要指出规约中最容易导致目标软件在解析时发生故障的位置。此类模糊器有 SPIKE 和 SPIKEfile，均以 SPIKE 脚本来描述目标协议或文件格式。

常用的模糊器有本地模糊器、远程模糊器、内存模糊器和模糊测试框架。本地模糊器主要用来生成本地测试目标的测试用例，通常会关注 3 类输入：命令行参数、环境变量参数和文件格式。远程模糊器是以监听一个网络接口的远程软件为测试目标。随着 Internet 的发展，几乎所有的公司都会公开一些可被访问的服务器，以提供 Web 页、E-mail、域名系统(DNS)解析服务器等，这类软件中的漏洞为攻击者提供了访问敏感数据或对更可信的服务器发起攻击的机会。远程模糊器包括网络协议模糊器、Web 应用模糊器和 Web 浏览器模糊器。内存模糊器的实现方法是冻结进程并存储其快照，然后快速将可能造成故障的数据注入进程解析例程。模糊测试框架是一个通用的模糊器或模糊器库，它简化了许多不同类型的测试目标的数据表示，例如 SPIKE 和 Peach。典型的模糊测试框架包括 3 个部分：一个漏洞触发方法库，用来产生模糊测试字符串或者通常能够导致解析例程出现问题的数据值；一个例程，以简化网络和磁盘输入输出；某种脚本类语言(如 Python)，在创建具体的模糊器时使用。模糊测试框架具有可重用性，需要根据测试目标的特点选择使用。

4) 执行模糊测试用例

执行模糊测试就是将上一阶段生成的大量模糊测试数据不断发送给待测目标程序。

面对大量的模糊测试数据，同样需要使用自动化工具来完成执行。因此，自动化对于模糊测试非常重要，没有自动化就无法真正地执行模糊测试。自动化模糊测试流程如图 7-7 所示。

5）监视异常和故障

监视异常和故障是模糊测试中至关重要的一步，必须记录是哪个用例引起了待测目标的异常和故障，这样才能通过重放该用例或之前的一组用例对异常和故障进行分析。如果某个用例引起了待测目标的崩溃，监视器有时还需要有恢复待测目标状态的功能。

6）检测可被利用的漏洞

在发现软件故障后，由于测试目标的不同，需要判断发现的故障是否是一个可被利用的安全漏洞。这种判定过程是典型的手工过程，需要执行人员具有安全领域的专业知识。

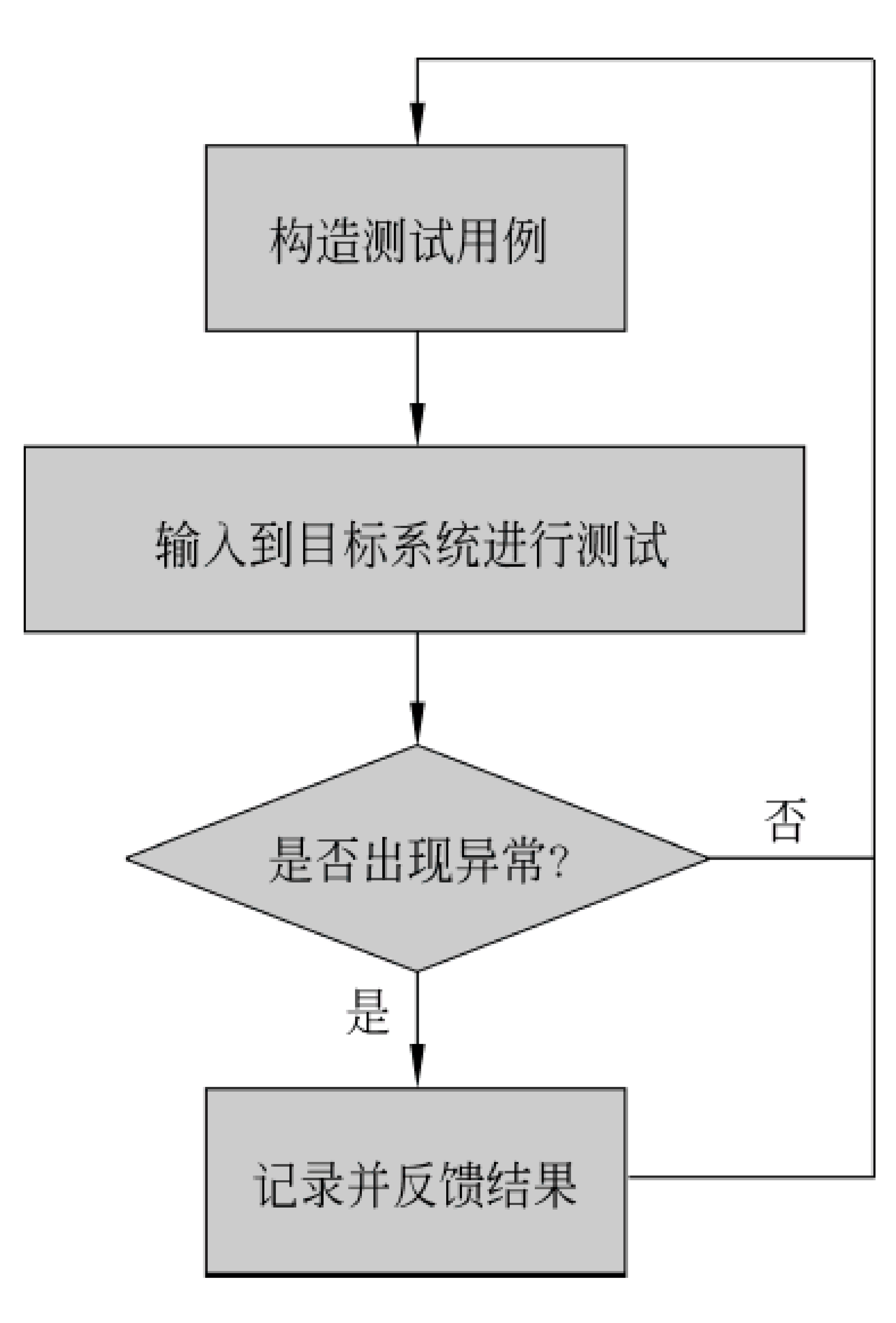


图 7-7 模糊测试工作流程

7.3.4 渗透测试

渗透测试通过模拟恶意黑客的攻击方法来评估系统的安全状况，该过程包括对系统弱点、技术缺陷或漏洞的分析。渗透测试具有两个显著的特点：渗透测试是一个逐步深入的过程；测试中一般不会对业务系统的正常运行造成影响。渗透测试发现的问题都是客观存在的，也较为严重，但是它只能覆盖有限的测试点。渗透测试可以使用自动化测试工具，同时为提升测试效果，测试人员需要有专业技能和丰富的经验，了解是什么导致软件安全和不安全，如何像攻击者一样思考，如何使用不同测试工具和技术模仿攻击者行为。值得注意的是：渗透测试主要模拟真实场景，分析入侵者可能的攻击路径，因此，需要系统部署完成后才能进行测试。渗透测试对象包括操作系统、数据库、应用系统和网络设备 4 种类型，如表 7-3 所示。

表 7-3 渗透测试对象及主要目标

测试对象	主 要 目 标
操作系统	Windows、Solaris、AIX、Linux、SCO、SGI 等
数据库	MS-SQL、Oracle、MySQL、Informix、Sybase、DB2、Access 等
应用系统	各种应用软件或程序，如 ASP、CGI、JSP、PHP 等组成的 Web 应用程序
网络设备	防火墙、入侵检测系统等

随着安全业界看待和定义渗透测试过程的方式的转变，安全业界中多个领军企业所采纳的渗透测试执行标准(Penetration Testing Execution Standard，PTES)对渗透测试进行了重新定义。新标准的核心理念是通过建立进行渗透测试所要求的基本准则基线来

定义一次真正的渗透测试过程。这一理念得到安全业界广泛认同。PTES 标准将渗透测试过程划分为如图 7-8 所示的 7 个阶段：前期交互、情报收集、威胁建模、漏洞分析、渗透攻击、后渗透攻击和报告。每个阶段中定义了不同的扩展级别，而选择何种级别则由被攻击测试的客户组织决定。与此同时，为使渗透测试更加有效，最优方法是将滥用用例、威胁建模等作为渗透测试活动的基础，用风险管理的思想来指导测试。

前期交互 → 情报收集 → 威胁建模 → 漏洞分析 → 渗透攻击 → 后渗透攻击 → 报告

图 7-8　PTES 标准渗透测试过程

1. 前期交互阶段

前期交互阶段通常是与客户组织进行讨论，以确定渗透测试的范围和目标。这个阶段最为关键，需要让客户组织清晰地了解渗透测试将涉及哪些目标，从而选择更加现实可行的渗透测试目标。

2. 情报收集阶段

在情报收集阶段，需要使用各种方法来收集目标的信息，包括使用社交媒体等网络信息范围内的已知事物、Google Hacking 技术、目标系统踩点等。作为渗透测试人员，最重要的一项技术就是对目标系统的探查能力，包括获知它的行为模式、运行机理以及最终可以如何对它进行攻击。

情报收集的主要内容是系统、网络和应用相关信息。系统信息包括系统的“旗标”、存在的漏洞等，可使用端口扫描和服务扫描获得。网络信息包括域名、网络结构等。应用信息主要是应用服务器的版本、平台信息和已发布的漏洞等。

在情报收集阶段，可通过逐步深入的探测来确定目标系统中实施的安全防御措施。例如，一个组织在对外开放的网络设备上经常设置端口过滤，只允许接收发往特定端口的网络流量，而一旦在白名单之外的端口访问这些设备，就会被加入黑名单以进行阻断。实现这种阻断行为的一个方法是从测试者所控制的其他 IP 地址开始进行初始探测，而这个 IP 地址是预期就会被阻断或者被检测到的。

在收集到信息之后，还需要对信息进行整理和分类，从而提取出对渗透测试有用的信息。

3. 威胁建模阶段

威胁建模主要使用在情报收集阶段所获取的信息标识出目标系统上可能存在的安全漏洞与弱点。在进行威胁建模时，要确定最高效的攻击方法、需要进一步获取的信息以及从哪里攻破目标系统。在威胁建模阶段，需要测试人员以攻击者的视角和思维来尝试利用目标系统的弱点。

4. 漏洞分析阶段

在漏洞分析阶段，需综合前 3 个阶段中获取的信息，并从中分析和确定可行、有效的攻击路径和方法。特别是需要重点分析端口和漏洞扫描结果、获取的服务“旗帜”信息以及在情报收集阶段中得到的其他关键信息。同时，在确定攻击路径后，还要分析如何获取

目标系统的访问控制权。

5. 渗透攻击阶段

渗透攻击阶段是真正入侵到目标系统中，获得访问控制权。在实际情况下，渗透攻击往往没有所预想的那么“一帆风顺”，而是“曲径通幽”。最好是在基本确信特定的渗透攻击会成功的时候，才真正对目标系统实施渗透攻击。当然，在目标系统中很可能存在着一些没有预料到的安全防护措施，使得这次渗透攻击无法成功。但需要注意的是，在尝试触发一个漏洞时，应该充分了解目标系统和利用漏洞。漫无目的的尝试只会造成大量的报警信息，并不会为测试人员和客户组织提供任何帮助。需先做好准备，然后再针对目标系统实施已经深入研究和测试的渗透攻击，这样才有可能取得成功。

6. 后渗透攻击阶段

后渗透攻击阶段从入侵到目标系统中或取得域管理权限之后开始。后渗透攻击阶段将以特定的业务系统为目标，识别出关键的基础设施，并寻找客户组织最具价值和尝试进行安全保护的信息和资产，当从一个系统攻入另一个系统时，需要演示出能够对客户组织造成最重要业务影响的攻击途径。在后渗透攻击阶段对系统进行攻击时，需要投入更多的时间来确定各种不同系统的用途以及它们不同的用户角色。

7. 报告阶段

报告是渗透测试过程中最为重要的因素，在报告文档中需要体现测试人员在测试过程中的测试内容、测试过程，最为重要的是，还要就客户组织应该如何修复测试所发现的安全漏洞和弱点提出建议。在此阶段，测试人员需要从客户组织的角度来分析如何利用发现的问题提升安全意识，修补发现的问题，以及提升系统整体的安全水平，而不仅仅是给发现的安全漏洞打上补丁。

报告至少分为摘要、过程展示和技术发现 3 个部分。技术发现部分将会被客户组织用来修补安全漏洞，这也是渗透测试过程真正价值的体现。例如，在客户组织的系统中发现了一个 SQL 注入漏洞，渗透测试人员将会在报告的技术发现部分建议客户组织对所有的用户输入进行检查过滤，使用参数化的 SQL 查询语句，在一个受限的用户账户上运行 SQL 语句，以及使用定制的出错信息。需要注意，最可能导致 SQL 注入漏洞的原因是使用了未能确保安全性的第三方应用，在报告中也要充分考虑这些因素，并建议客户组织进行细致的检查，进而消除这些漏洞。

第8章 软件安全发布与部署

互联网技术的发展使软件成为人们现实生活中做很多事情的关键,同时,软件也分布在各个关键系统之中。因此,软件安全已经成为人们关注的重点。软件安全指在软件开发过程中应用安全开发周期管理模式构建安全的软件。虽然在软件开发过程中较早修复安全缺陷将比发布后再修复安全漏洞大大节约成本,但是我们仍需关注软件开发生命周期各阶段的安全审查。目前,大多数信息技术相关的安全措施已经能够有效提高软件安全性,降低软件安全风险。本章主要介绍软件发布以及部署阶段相关的安全概念以及安全措施。

8.1 软件安全发布

软件发布指交付软件产品,在该阶段仍需对软件进行安全性检查,以确保软件是安全的,即安全问题已经被解决到软件发布时可以接受的程度。下面对最终安全性检查以及安全事故响应计划进行详细阐述。

8.1.1 最终安全审查

安全审查在软件安全开发生命周期各阶段都不容忽视,因此,在软件开发的发布阶段,需对软件进行最终的安全审查,以确保软件产品已经为发布做好所有准备。在最终的软件开发安全审查过程中,所有的安全活动,例如威胁模型、测试输出、性能需求等均需要进行确认,并对其进行重新评估。通过最终安全审查,可能出现的结果有3种,分别为:最终安全审查通过;最终安全审查通过,但是有异议;最终安全审查没有通过,需要进行扩展。接下来将对这3种情况进行详细描述。

(1) 最终安全评审通过。

该结果表明所有已经确认的最终安全问题都已经得到修正,并且已确认软件满足所有SDL需求。因此,从软件安全性角度来说已经达到软件发布要求。

(2) 最终安全审查通过,但是有异议。

该结果表明所有已经确认的安全问题并未得到全部修正,但对于开发团队无法处理的一个或多个异常给予接收处理。与此同时,此异常虽然不会在当前版本解决,但是会被定位在接下来的补丁或版本中进行修正。

(3) 最终安全审查没有通过,需要进行扩展。

此结果表明所有已经确认的安全问题并未得到全部修正,并且SDL和开发团队无法接受存在的安全漏洞和修补情况,因此,软件产品不能发布。导致软件无法发布的SDL

需求必须扩展至更高的管理层面进行决策，从而评估在不满足需求的情况下进行软件发布带来的风险与后果。在决策过程中，应该依据 SDL 和开发团队提交的报告，报告中包含安全风险描述及其理由。

从上述 3 个结果中可以看出，存在安全审查不通过的情况，此时，需要对已知问题进行修复。因此，最终安全审查必须制订时间表并严格执行，从而达到完全分析时间、修复已知问题时间、修复最终安全审查发现的安全问题时间最大化的目标，进而使得软件产品发布的时间更加充裕。

产品发布阶段的安全审查作为产品的最终安全审查，决定了软件产品能否发布。主要可分为 4 个步骤，如图 8-1 所示，分别为评估资源可用性、确定合格的特征、评估和制订修复计划以及版本发布。接下来将对这 4 个步骤的具体内容进行详细阐述。

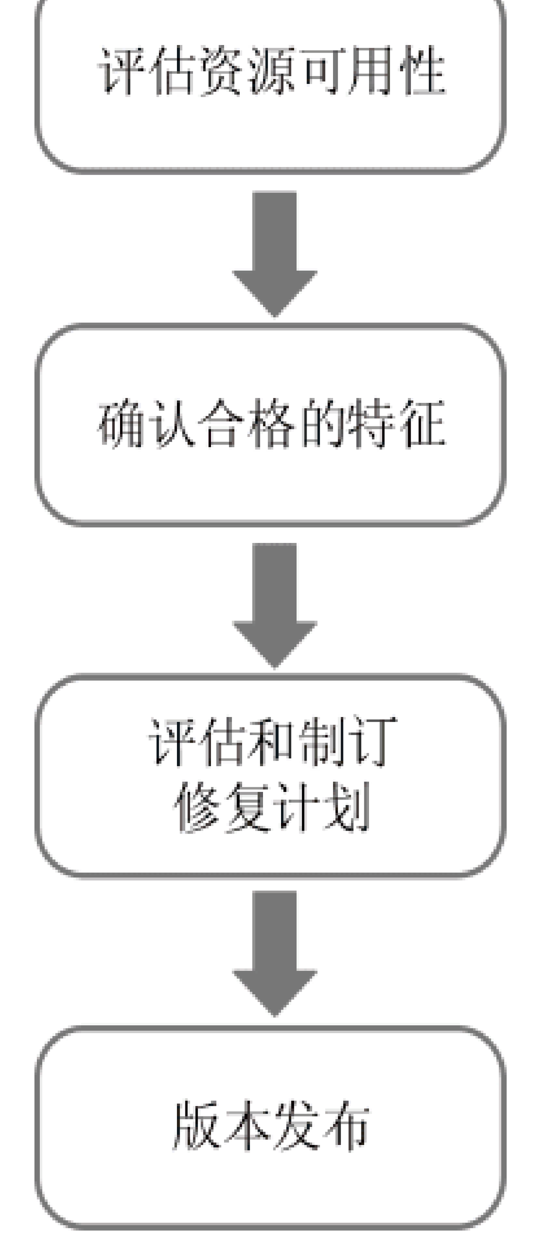


图 8-1 最终安全审查流程

（1）评估资源可用性。

评估资源可用性作为最终安全审查的第一步，主要确定执行最终安全审查的资源。与此同时，在软件发布前评估增强质量的能力标准，通过质量标准建立可接受的最低安全水平。其中，质量标准应该在软件安全开发生命周期的早期建立，从而使得安全风险能够较早被发现，进而保证安全漏洞能够在早期被确认和修复，避免不需要的工作导致软件发布延期。值得重视的是，在最终安全审查过程中，SDL 和开发团队必须遵守质量标准，如果安全已经确认作为 SDL 的结果在软件安全开发生命周期中得到建立，那么将最终安全审查时间最小化；反之，则需要投入过多的时间资源，甚至可能导致软件发布的延期。

（2）确认合格的特征。

确认合格的特征作为最终安全审查的第二步，需要确认最终安全审查的安全任务的合格标准。在软件开发过程早期建立合格特征，从而避免安全审查中包含未完成的安全任务。在软件开发过程中，可能存在没有报告的安全漏洞，而在最终安全审查过程中发现遗留的高风险安全漏洞。

（3）评估和制订修复计划。

评估和制订修复计划作为最终安全审查的第三步，需要通知任务的相关负责人，并且对最终安全审查的日程安排进行确认。在该步中，需评估所有安全活动，在必要时制订修复计划。

（4）版本发布。

产品安全审查需对模糊测试、安全漏洞扫描、安全编码原则检查等进行审查，以确保软件产品满足所有 SDL 需求，并且满足发布要求。其中，功能回归测试会占用安全审查的资源。回归测试用以发现新软件安全漏洞或根据已经发现的安全漏洞进行回归测试，值得注意的是，这些回归测试可能导致软件已经存在的功能性和非功能性方面发生变化。因此，回归测试主要评估软件的一部分发生变化时是否会导致软件中与其交互的部分也发生变化。

8.1.2　安全事故响应计划

无论软件在安全方面和相关的 SDL 方面做得多么好,都会忽略一些问题,为此需要制订安全事故响应计划。因此,产品发布后的管理关键就是要建立产品安全事故响应小组(Product Security Incident Response Team,PSIRT)。产品安全事故响应小组负责应对软件产品安全事件,涉及发布后软件产品安全漏洞的外部发现。安全漏洞的外部发现者可能是独立的安全研究人员、顾问、行业组织、其他厂商以及善意或者恶意的黑客,他们能够找出 PSIRT 负责的软件产品中可能存在的安全漏洞。

8.2　软件安全部署

软件部署指与软件配置、维护相关的过程、活动和措施。与此同时,软件部署也关注其他对软件的安全性有直接影响的环境问题。本节以软件保证成熟度模型的部署功能为例,详细介绍软件的安全部署。SAMM 设置了 4 种关键业务;对于每一个业务功能,SAMM 设置了 3 个安全措施;对于每一个安全措施,SAMM 设置了 3 个成熟度等级。其中,部署功能的 3 个安全措施分别为漏洞管理、环境强化和操作激活。接下来将分别对其进行详细介绍。

8.2.1　漏洞管理

漏洞管理(vulnerability management)功能注重关于处理漏洞报告和操作时间的组织内部流程。从发生事件时简单地分配角色入手,组织会逐步形成正式的事件响应流程,以确保能够对发生的问题进行跟踪。漏洞管理涉及对事件和报告的分析,并收集详细的衡量指标。漏洞管理的 3 个成熟度等级分别为 VM1、VM2、VM3,其具体内容可参阅 OWASP 的相关文档。

8.2.2　环境强化

环境强化(environment hardening)功能注重软件的运行环境,强化操作环境能够有效改进软件的整体安全状态。组织可以从简单跟踪和向开发团队分发有关操作环境的信息入手,逐步采纳可扩展的方法,管理安全补丁部署和使用早期警告检测器,以尽早发现潜在安全风险。随着组织的发展,还可以部署保护工具,添加防护层和安全网,从而限制漏洞被利用时造成的损失。环境强化分为 3 个成熟度等级,分别为 EH1、EH2、EH3,具体内容可参阅 OWASP 的相关文档。

8.2.3　操作激活

操作激活(operational enablement)功能注重从开发软件的项目团队收集重要的安全信息,并将其传达给软件的用户和操作人员。组织应从为用户和操作人员提供附有详细信息的简易文档入手,逐渐制定随每个软件版本提供的完整的操作安全指南。操作激活分为 3 个成熟度等级,分别为 OE1、OE2、OE3,具体内容可参阅 OWASP 的相关文档。

第9章 典型案例

本章介绍代码审计系统的应用实例,根据相应的应用背景和安全需求,分析其存在的安全问题,提出具有针对性的解决方案,并以奇安信网神代码卫士为例,展示其部署方式和方案效益。

9.1 应用背景

随着网络技术和应用的飞速发展,信息系统安全正面临着前所未有的挑战。近几年重大安全事件的频频发生表明了当前信息系统安全形势的严峻性,仅仅依靠传统的安全防护机制来保障信息安全的做法已经逐渐力不从心。软件代码是构建系统信息的基础,软件代码中安全漏洞和未声明功能(后门)的存在是安全事件频繁发生的根源。忽视软件代码自身的安全性,仅仅依靠外围的防护、事后的修补等方法,舍本逐末,必然事倍功半。只有通过管理和技术手段保障软件代码自身的安全性,再辅以各种安全防护手段,才是解决当前安全问题的根本解决之道。

9.2 企业需求

美国等西方发达国家非常重视软件代码安全保障,从政府部门到企业界都在积极推进这一工作。美国国土安全部提出软件"内建安全"的概念,将安全作为软件的基础属性,并资助了一系列软件代码安全保障的研究项目,如SAMATE、开源代码安全测试计划等;企业界则以微软公司为代表,提出了软件安全开发生命周期的理念,强调软件整个生命周期各个环节的安全保障,这一理念也已被众多大型企业所采纳。

某企业十分重视软件代码的安全,希望通过部署源代码安全检测产品来解决软件开发和测试过程中遇到的安全问题。但该企业在源代码安全检测技术应用中遇到了以下几个问题:

(1) 国外源代码检测产品占据市场主流。

软件源代码是企业的核心资产和重要知识产权。源代码安全检测产品的应用是否会引入其他的安全风险,如何保障源代码安全检测产品自身安全可控,是企业安全主管领导关心的核心问题。目前市场上的源代码安全检测产品大都是国外厂商开发的,相关实现原理极少对外公开,因此,对应用这些产品的企业来说,其自主可控性大打折扣。

（2）现有产品难以与开发或测试流程融合。

企业已经购买了一些源代码安全检测产品（如 Fortify、Checkmarx），但使用效果不理想，产品利用率不高。传统的软件开发和测试流程并未考虑源代码自身安全的需求。测评机构曾经尝试在已固化的流程中应用源代码安全检测产品，但遇到了各种具体问题，例如，源代码安全检测产品难以与代码管理服务器进行对接，无法与已有的缺陷管理系统自动进行结果整合，等等。这些问题如果不能得到有效解决，将会大大地增加开发和测试工作量，影响工作效率。

（3）多数产品对个性化需求的支持不足。

当前企业希望建设一个检测管理平台，以整合多个源代码检测工具，同时对多个源代码安全检测产品的结果进行深度融合，提供统一的检测报告。目前常见的商业源代码安全检测产品大多是根据通用需求开发的，但不同的企业由于行业及业务的特殊性，对源代码安全检测有很多个性化的合规性需求。目前的商业检测工具对这种个性化需求的支持不足，导致组织在应用源代码安全检测产品时无法满足自身的需求。

（4）开发过程中较少评估开源组件的安全风险。

开源组件存在的安全漏洞和授权协议问题会给软件的使用带来风险。个别研发人员为了方便，直接引用来自互联网的开源组件进行编码，这些开源组件往往包含安全漏洞，由此降低了软件的整体安全性。另外，开源组件一般都需获得相关的使用授权，直接引用这些开源模块可能会给企业带来法律风险。目前测评机构还未实施能够对开源代码进行溯源评估的有效技术措施。

9.3 解决方案

为解决测评机构在源代码安全检测方面遇到的问题，奇安信集团结合多年源代码安全检测及审计经验，为企业部署代码安全保障系统，根据企业已经购买过的安全产品进行定制化开发融合，为企业建设一体化的源代码安全保障体系。

为了建立实用、可靠的源代码安全保障体系，应首先建立明确的规则，然后搭建高效的检测平台，并对研发人员进行培训，使其具备扎实的安全技能，最终辅以安全专家的深度安全分析服务。源代码安全保障体系建设思路如图 9-1 所示。

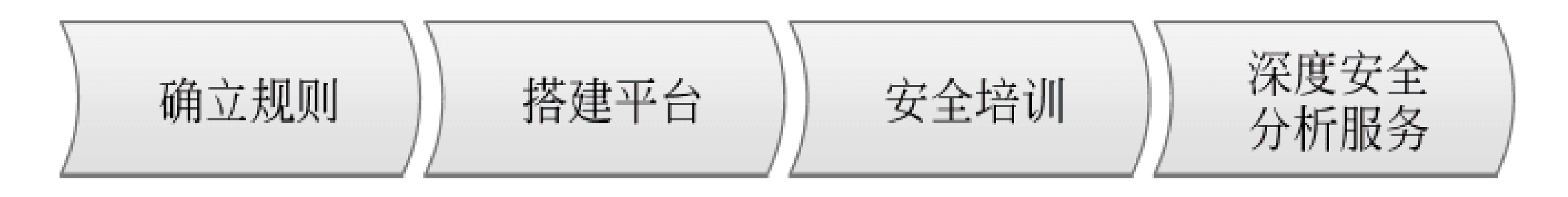


图 9-1　源代码安全保障体系建设思路

1. 确立规则

在确立规则阶段，资深代码安全专家将分析测评机构已有的开发与测试流程，确立代码安全保障工作流程，制订流程整合方案。分析企业相关业务系统源代码特点与历史安全状况，总结梳理企业系统源代码中经常出现的安全问题，确立代码安全目标，并结合国际主流标准，针对性地制定符合企业特点的安全开发与测试规范，进而开发自动化缺陷及

合规检测规则集，支撑源代码安全检测的流程自动化。

2.搭建平台

在搭建平台阶段，基于代码安全保障系统搭建自动化检测平台。代码安全保障系统是奇安信集团开发的基于软件安全开发生命周期管理的新一代源代码安全检测系统。它面向企业的源代码安全需求，能够在不改变企业现有开发流程的前提下，帮助企业实现使用一个检测管理平台整合多个源代码检测工具，对多个源代码安全检测产品的结果进行深度融合的目标。

根据测评机构现状，奇安信集团建议采用一套代码安全专用硬件进行部署。该专用硬件包含了代码安全管理中心、服务交换机及检测引擎等相关模块。企业无须进行额外的软件安装部署，仅需安装代码安全检测系统，即可开始源代码安全检测工作。

上传到源代码安全检测平台的源代码通过服务交换机后，由任务调度子系统分配代码检测方式和工具；检测结果在返回服务交换机后，又通过结果综合处理、结果对应与转换、结果去重等处理，获得综合的分析处理结果；最后将原始代码信息、检测出的缺陷漏洞和处理后的结果分别存入用户信息库、缺陷知识库和分析结果库中，以方便以后的查询分析。代码安全检测系统结构如图 9-2 所示。

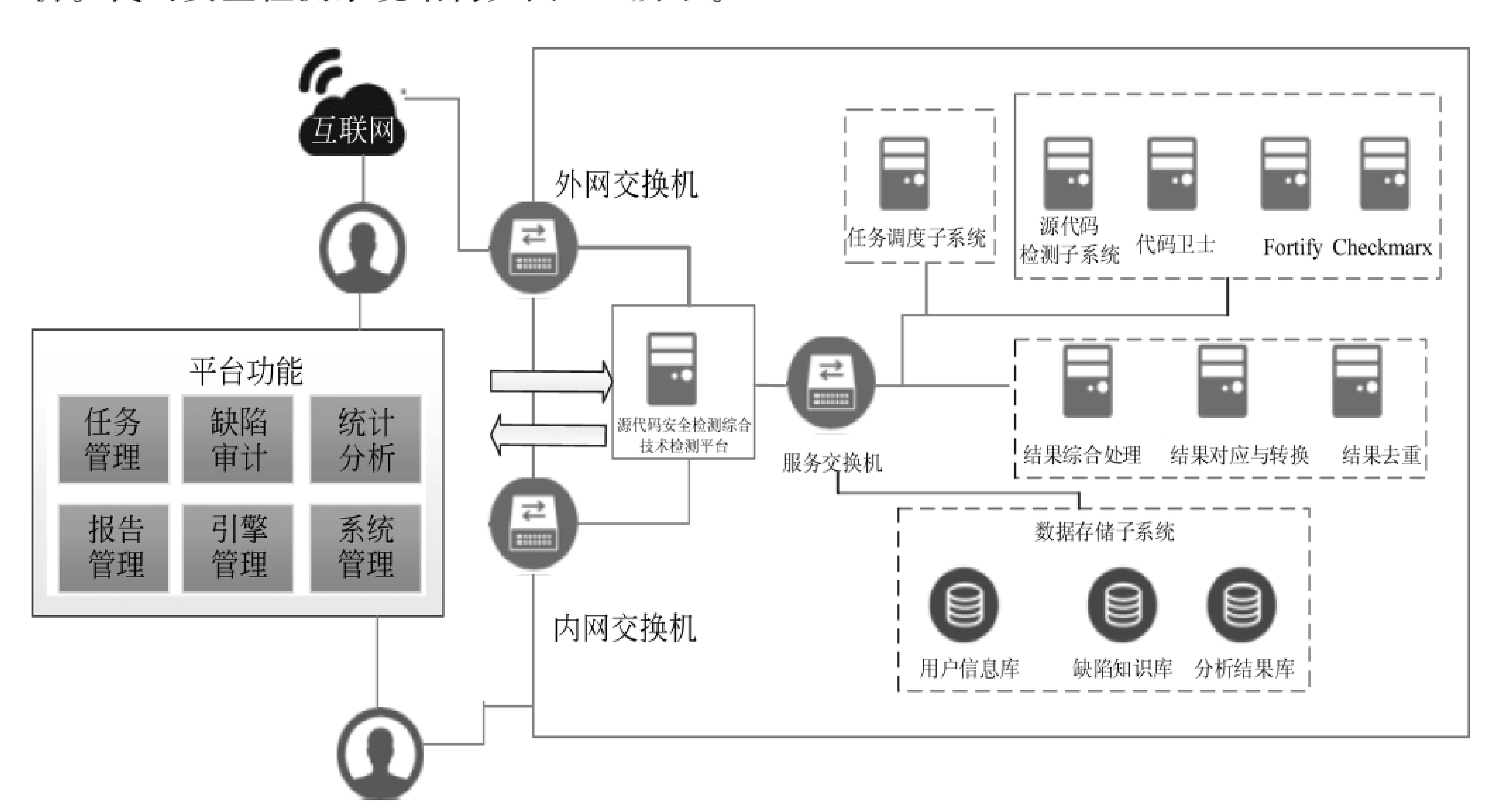


图 9-2　代码安全检测系统结构

3.安全培训

在安全培训阶段，通过培训，使软件设计、开发、测试等相关人员能够了解软件安全开发生命周期在软件开发过程中的重要作用，了解 Java、C/C++、C# 等语言的安全编码常识，了解常见的源代码安全缺陷及其产生机理、严重危害和防范措施。

4.深度安全分析服务

在深度安全分析服务阶段，由资深代码安全专家提供深度安全分析服务、缺陷成因分析以及修复建议。

9.4 方案优势

奇安信集团的代码审计安全解决方案具有以下 5 个优势：

(1) 有效降低软件安全修复成本。

源代码安全检测是缓解软件安全问题的有效途径，可从源头上大量减少代码注入、跨站脚本等高危安全问题，进而提升信息系统的安全性。根据 Gartner 公司的统计，在软件代码实现阶段发现并纠正安全问题所花费的成本是软件交付后通过“上线安全评估”发现问题再进行整改的成本的 1/1000～1/50。越早发现源代码安全问题，其修复成本越低。代码安全检测系统可帮助企业在软件开发过程中尽早、尽快发现软件源代码安全问题，有效降低软件修复成本。

(2) 自主可控的源代码安全检测。

软件源代码是企业的核心信息化资产。源代码安全检测产品部署到开发和测试网络中之后是否会引入其他的安全风险，如何保障源代码安全检测产品自身安全可控，是企业最关心的问题。代码安全检测系统是奇安信集团自主研发的国产源代码安全检测产品，该解决方案符合国家对信息安全产品自主、可控的要求。

(3) 自主化的安全开发生命周期管理。

代码安全检测系统在设计时充分考虑了企业在应用源代码安全检测产品过程中遇到的各种实际问题，其基本理念是使企业用最小的代价将源代码安全检测操作融入已有开发和测试流程中，实现安全开发生命周期管理。代码安全检测系统支持软件项目安全目标设定，可以从 SVN、GIT 等代码库获取源代码并进行自动化周期检测，支持检测结果差距分析、趋势分析，检测结果可与 Bugzilla 等缺陷管理系统进行整合，企业无须改变原有流程，即可享有软件源代码质量大幅度提升带来的好处。

(4) 支持多方面个性化定制开发。

相对于国外品牌，代码安全检测系统更贴近企业的需求。奇安信集团可根据企业的行业特性及业务特点，提供个性化的源代码安全检测规范定制开发，满足信息安全等级保护三级系统对于自主开发软件“应制定代码编写安全规范，要求开发人员参照规范编写代码”的要求。同时，针对企业已经购买的 Foritify、Checkmarx 等商业检测工具，代码安全检测系统提供了定制化接口，可驱动企业已有工具对软件源代码进行复检，并统一反馈检测结果，既利用互补优化了源代码检测结果，也保护了企业原有投资。

(5) 该方案是国内唯一的开源代码风险评估产品。

代码安全检测系统是国内唯一支持溯源检测的开源代码自动化检测产品。开源组件存在的安全漏洞和授权协议问题会给企业带来风险，代码安全检测系统能够检测软件中使用了哪些开源组件，这些组件存在哪些安全漏洞和使用授权的问题，帮助企业降低使用开源代码的风险。

附录A 英文缩略语

AOP　Aspect-Oriented Programming　面向方向的编程

AOSD　Aspect-Oriented Software Development　面向方向的软件开发

API　Application Programming Interface　应用编程接口

BSIMM　Building Security In Maturity Model　BSI成熟度模型

CC　Common Criteria for Information Technology Security Evaluation　信息技术安全评估通用标准

CLASP　Comprehensive Lightweight Application Security Process　综合的轻量级应用安全过程

DBA　Database Administrator　数据库管理员

DSA　Digital Signature Algorithm　数字签名算法

DSS　Digital Signature Standard　数字签名标准

EH　Environment Hardening　环境强化

JLS　Java Language Specification　Java语言规范

OE　Operational Enablement　操作激活

OWASP　Open Web Application Security Project　开放Web应用安全项目

PHP　Hypertext Preprocessor　超级文本预处理语言

PSIRT　Product Security Incident Response Team　产品安全事故响应小组

PTES　Penetration Testing Execution Standard　渗透测试执行标准

RAI　Resource Acquisition Initialization　资源获取初始化

RUP　Rational Unified Process　统一软件开发过程

SAMM　Software Assurance Maturity Model　软件保证成熟度模型

SDL　Security Development Lifecycle　安全开发生命周期

SDLC　Software Development Lifecycle　软件开发生命周期

SHA　Secure Hash Algorithm　安全哈希算法

SQUARE　Security QUAlity Requirements Engineering　安全质量需求工程

TOE　Target Of Evaluation　分析评估对象

VM　Vulnerability Management　漏洞管理

XML　eXtensible Markup Language　可扩展标记语言

参考文献

[1] 吴世忠. 软件安全开发[M]. 北京：机械工业出版社，2016.

[2] Long F，Dhruv M. Java安全编码标准[M]. 计文柯，杨晓春，译. 北京：机械工业出版社，2013.

[3] Seacord R C. C和C++安全编码[M]. 卢涛，译. 北京：机械工业出版社，2014.

[4] 尹毅. 代码审计[M]. 北京：机械工业出版社，2015.

[5] 黄正鹏. Java语言的程序漏洞分析技术研究[J]. 电脑编程技巧与维护，2016，12(2)：21-22.

[6] 陈华. Java Web应用程序安全技术研究[J]. 电脑编程技巧与维护，2010，20(24)：123-124.

[7] 刘鹏. PHP Web应用程序安全性研究及安全漏洞检测工具开发[D]. 西安：西安电子科技大学，2011.

[8] 周璐. 一种PHP程序自动化缺陷分析工具的设计与开发[D]. 成都：电子科技大学，2014.

[9] 张实君. phpBB安全漏洞的分析及高级逃逸技术的研究[D]. 北京：华北电力大学，2016.

[10] 宋明秋. 软件安全开发——属性驱动模式[M]. 北京：电子工业出版社，2016.

[11] Visser J. 代码不朽：编写可维护软件的10大要则[M]. 张若飞，译. 北京：电子工业出版社，2016.

[12] Martin R C. 代码整洁之道：程序员的职业素养[M]. 余晟，章显洲，译. 北京：人民邮电出版社，2016.

[13] 魏家明. 代码结构[M]. 北京：电子工业出版社，2016.

[14] 詹姆斯·兰萨姆，安莫尔·米斯拉. 软件安全：从源头开始[M]. 丁丽萍，卢国庆，李彦峰，译. 北京：机械工业出版社，2016.

[15] 彭国军，傅建明，梁玉编，等. 软件安全[M]. 武汉：武汉大学出版社，2015.

[16] 姜文，刘立康. C++与Java软件重量级静态检查[J]. 计算机技术与发展，2016，26(8)：17-23.

[17] 冯博. 软件安全开发关键技术的研究和实现[D]. 北京：北京邮电大学，2010.

[18] 白哥乐，宫云战，杨朝红，等. 基于源码分析的软件安全测试工具综述[C]//中国计算机学会. 第五届中国测试学术会议论文集. 中国计算机学会，2008.

[19] 徐甜. Java平台及应用Java技术的安全问题研究[J]. 微计算机信息，2007，23(18)：216-218.

[20] 陆钟石. Java安全体系结构设计与实现[D]. 北京：北京邮电大学，2010.

[21] 张亚林，王开磊. Java Web应用程序安全技术研究[J]. 计算机光盘软件与应用，2012(4)：58-61.

[22] 苟有来. Java安全类在企业信息系统中的应用研究[D]. 北京：北京林业大学，2005.

[23] 程茂华. 基于PHP安全漏洞的Web攻击防范研究[J]. 信息安全与技术，2013，4(5)：53-55.

[24] 程茂华. PHP安全漏洞防范研究[J]. 信息安全与技术，2013，4(7)：75-77.

[25] 刘鹏. PHP Web应用程序安全性研究及安全漏洞检测工具开发[D]. 西安：西安电子科技大学，2011.

[26] 肖胜仁. 浅析PHP安全漏洞防范技术分析[J]. 网络安全技术与应用，2015(4)：123-127.

[27] 林龙成. PHP Web应用程序开发中漏洞消减技术研究[D]. 南京：南京师范大学，2014.

[28] 王晓华. 软件安全测试方法研究[J]. 农业网络信息，2010(3)：124-128.

[29] 黄奕. 基于模糊测试的软件安全漏洞发掘技术研究[D]. 合肥：中国科学技术大学，2010.

[30] 张蕾. 软件安全测试技术和工具的研究[J]. 中国新技术新产品，2017(17)：21-22.

[31] 张志海，郑春一，张红军，等. 一种面向等级保护的软件安全需求分析方法研究[J]. 信息网络安

全,2013(8):2-4.
[32] 吕维梅,刘坚. C/C++程序安全漏洞的分类与分析[J]. 计算机工程与应用,2005,41(5):123-125.
[33] Fisher G. C、C++和Java安全编码实践提示与技巧[J]. 程序员,2009(2):89-92.
[34] 胡忠帅. 基于Python的企业安全漏洞管理方法研究[D]. 北京:北京邮电大学,2014.
[35] 叶磊,文涛,刘立亮,等. 基于Python的网络及信息系统安全过程管理工具[J]. 数字技术与应用,2017(10):187-188.
[36] 褚诚云. 安全编码实践三:C/C++静态代码分析工具Prefast[J]. 程序员,2008(6):109-111.
[37] 龙刚,何建安,高嵩,等. C/C++源代码安全检测系统的设计与实现[J]. 数字技术与应用,2016(6):150-151.
[38] 王泓,李洪敏. 防止缓冲区溢出——C/C++语言的安全问题[C]//中国计算机学会.第十九次全国计算机安全学术交流会论文集.中国计算机学会,2006.
[39] 王世华,沈卫超. 用Python和wxPython开发主机安全监控系统[C]//中国计算机学会.第二十次全国计算机安全学术交流会.中国计算机学会,2008.
[40] 王若宇. 一种Python程序模块的安全调用方法和装置[J]. 北京:华青融天(北京)技术股份有限公司,2016.
[41] 孙明. 基于虚拟机的软件动态分析方法研究[D]. 上海:上海交通大学,2011.
[42] 彭长艳. 空间网络安全关键技术研究[D]. 长沙:国防科学技术大学,2010.
[43] 何可. 威胁模型驱动的软件安全评估与测试方法的研究[D]. 天津:天津大学,2010.
[44] 李洪波. 基于安全需求模板的软件安全需求获取工具设计与实现[D]. 天津:天津大学,2016.
[45] 王涛. 基于安全模式的软件安全设计方法[D]. 长春:吉林大学,2011.
[46] 吴晓菲. 基于等级和形式化建模的软件安全需求自动获取方法与工具[D]. 天津:天津大学,2014.
[47] 李汶娟. 软件安全关注点建模方法研究[D]. 乌鲁木齐:新疆大学,2014.
[48] 王宏阳. 基于漏洞分析的软件综合检测方法研究[D]. 大连:大连海事大学,2016.
[49] 李晓南. 基于数据综合分析的软件安全漏洞静态检测平台设计与实现[D]. 成都:电子科技大学,2011.
[50] 朱明悦. 基于缺陷检测的软件安全风险评估工具设计与实现[D]. 天津:天津大学,2014.
[51] 吕一平. 软件安全定量分析和深层防御的应用研究[D]. 上海:上海交通大学,2007.